MC68000UM(AD)

16-BIT MICROPROCESSOR

User's Manual

MOTOROLA

MOTOROLA

MC68000

16-BIT MICROPROCESSOR
User's Manual

Original Issue: September 1, 1979

PRELIMINARY

©MOTOROLA INC., 1979

IMPORTANT NOTICE

This manual provides complete documentation for the **MC68000** microprocessor. Engineering prototype microprocessors are designated **XC68000**. Differences between the MC68000 and the XC68000 are covered in the XC68000 Exception Sheet accompanying the device.

This manual is limited to information on the MC68000 **microprocessor**. It does not define a specific assembly programming language.
The language and notation used within this manual is a register transfer language explained in Appendix B.

Motorola reserves the right to make changes to any products herein to improve functioning or design. Although the information in this document has been carefully reviewed and is believed to be reliable, Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others.

TABLE OF CONTENTS

Paragraph No.	Title	Page No.
	Chapter 1 General Description	
1.0	Introduction	1-1
1.1	Overview	1-1
1.2	Software Development	1-8
1.2.1	Consistent Structure	1-8
1.2.2	Structured Modular Programming	1-8
1.2.3	Improved Software Testability	1-9
1.2.4	Future Flexibility	1-10
1.2.5	Memory Management of Large Addressing Space	1-11
1.2.6	Improved Code Density and Speed	1-12
1.2.7	Software Support and M6800 Compatibility	1-13
	Chapter 2 Data Organization and Addressing Capabilities	
2.0	Introduction	2-1
2.1	Operand Size	2-1
2.2	Data Organization in Registers	2-1
2.2.1	Data Registers	2-1
2.2.2	Address Registers	2-1
2.3	Data Organization in Memory	2-2
2.4	Addressing	2-3
2.5	Instruction Format	2-4
2.6	Program/Data References	2-4
2.7	Register Notation	2-4
2.8	Address Register Indirect Notation	2-4
2.9	Register Specification	2-5
2.10	Effective Address	2-5
2.10.1	Register Direct Modes	2-5
2.10.1.1	Data Register Direct	2-5
2.10.1.2	Address Register Direct	2-6
2.10.2	Memory Address Modes	2-6
2.10.2.1	Address Register Indirect	2-6
2.10.2.2	Address Register Indirect with Postincrement	2-6
2.10.2.3	Address Register Indirect with Predecrement	2-7
2.10.2.4	Address Register Indirect with Displacement	2-7
2.10.2.5	Address Register Indirect with Index	2-8
2.10.3	Special Address Modes	2-9
2.10.3.1	Absolute Short Address	2-9
2.10.3.2	Absolute Long Address	2-9
2.10.3.3	Program Counter with Displacement	2-10
2.10.3.4	Program Counter with Index	2-11
2.10.3.5	Immediate Data	2-11
2.10.3.6	Condition Codes or Status Register	2-12

Table of Contents (Continued)

Paragraph No.	Title	Page No.
2.10.4	Effective Address Encoding Summary	2-12
2.11	Implicit Reference	2-13
2.12	Stacks and Queues	2-13
2.12.1	System Stack	2-13
2.12.2	User Stacks	2-14
2.12.3	Queues	2.15

Chapter 3
Instruction Set Summary

3.0	Introduction	3-1
3.1	Data Movement Operations	3-1
3.2	Integer Arithmetic Operations	3-2
3.3	Logical Operations	3-3
3.4	Shift and Rotate Operations	3-4
3.5	Bit Manipulation Operations	3-5
3.6	Binary Coded Decimal Operations	3-5
3.7	Program Control Operations	3-6
3.8	System Control Operations	3-7

Chapter 4
Signal and Bus Operation Description

4.0	Introduction	4-1
4.1	Signal Description	4-1
4.1.1	Address Bus (A1 through A23)	4-2
4.1.2	Data Bus (D0 through D15)	4-2
4.1.3	Asynchronous Bus Control	4-2
4.1.3.1	Address Strobe ($\overline{AS}$)	4-2
4.1.3.2	Read/Write (R/$\overline{W}$)	4-2
4.1.3.3	Upper and Lower Data Strobes ($\overline{UDS}$, $\overline{LDS}$)	4-2
4.1.3.4	Data Transfer Acknowledge ($\overline{DTACK}$)	4-3
4.1.4	Bus Arbitration Control	4-3
4.1.4.1	Bus Request ($\overline{BR}$)	4-3
4.1.4.2	Bus Grant ($\overline{BG}$)	4-3
4.1.4.3	Bus Grant Acknowledge ($\overline{BGACK}$)	4-3
4.1.5	Interrupt Control ($\overline{IPL0}$, $\overline{IPL1}$, $\overline{IPL2}$)	4-3
4.1.6	System Control	4-3
4.1.6.1	Bus Error ($\overline{BERR}$)	4-4
4.1.6.2	Reset ($\overline{RESET}$)	4-4
4.1.6.3	Halt ($\overline{HALT}$)	4-4
4.1.7	MC6800 Peripheral Control	4-4
4.1.7.1	Enable (E)	4-5
4.1.7.2	Valid Peripheral Address ($\overline{VPA}$)	4-5
4.1.7.3	Valid Memory Address ($\overline{VMA}$)	4-5

Table of Contents (Continued)

Paragraph No.	Title	Page No.
4.1.8	Processor Status (FC0, FC1, FC2)	4-5
4.1.9	Clock (CLK)	4-5
4.1.10	Signal Summary	4-6
4.2	Bus Operation	4-6
4.2.1	Data Transfer Operations	4-6
4.2.1.1	Read Cycle	4-7
4.2.1.2	Write Cycle	4-11
4.2.1.3	Read-Modify-Write Cycle	4-15
4.2.2	Bus Arbitration	4-18
4.2.2.1	Requesting the Bus	4-18
4.2.2.2	Receiving the Bus Grant	4-20
4.2.2.3	Acknowledgement of Mastership	4-20
4.2.3	Bus Error and Halt Operation	4-21
4.2.3.1	Exception Sequence	4-21
4.2.3.2	Re Running the Bus Cycle	4-22
4.2.3.3	Halt Operation with no Bus Error	4-23
4.2.3.4	Double Bus Faults	4-24
4.2.4	Reset Operations	4-25

Chapter 5
Exception Processing

5.0	Introduction	5-1
5.1	Processing States	5-1
5.2	Privilege States	5-1
5.2.1	Supervisor State	5-2
5.2.2	User State	5-2
5.2.3	Privilege State Changes	5-3
5.2.4	Reference Classification	5-3
5.3	Exception Processing	5-4
5.3.1	Exception Vectors	5-4
5.3.2	Kinds of Exceptions	5-6
5.3.3	Exception Processing Sequence	5-6
5.3.4	Multiple Exceptions	5-6
5.4	Exception Processing Detailed Discussion	5-7
5.4.1	Reset	5-7
5.4.2	Interrupts	5-8
5.4.3	Instruction Traps	5-10
5.4.4	Illegal and Unimplemented Instructions	5-11
5.4.5	Privilege Instructions	5-11
5.4.6	Tracing	5-12
5.4.7	Bus Error	5-12
5.4.8	Address Error	5-13

Table of Contents (Continued)

Paragraph No.	Title	Page No.

Chapter 6
Interface with M6800 Peripherals

6.0	Introduction	6-1
6.1	Data Transfer Operation	6-1
6.2	Interrupt Interface Operation	6-6

Appendix A
Condition Codes Computation

A.0	Introduction	A-1
A.1	Condition Code Register	A-1
A.2	Condition Code Register Notation	A-2
A.3	Condition Code Computation	A-2
A.4	Conditional Tests	A-4

Appendix B
Instruction Set Details

B.0	Introduction	B-1
B.1	Addressing Categories	B-1
B.2	Instruction Description	B-2
B.3	Register Transfer Language Definition	B-3
	Instructions	
	ABCD	B-4
	ADD	B-5
	ADDA	B-6
	ADDI	B-7
	ADDQ	B-8
	ADDX	B-9
	AND	B-11
	ANDI	B-12
	ASL, ASR	B-13
	Bcc	B-15
	BCHG	B-17
	BCLR	B-19
	BRA	B-21
	BSET	B-22
	BSR	B-24
	BTST	B-25
	CHK	B-26
	CLR	B-27
	CMP	B-28
	CMPA	B-29
	CMPI	B-30
	CMPM	B-31

Table of Contents (Continued)

Paragraph No.	Title	Page No.
	DB_{cc}	B-32
	DIVS	B-33
	DIVU	B-34
	EOR	B-35
	EORI	B-36
	EXG	B-37
	EXT	B-38
	JMP	B-39
	JSR	B-40
	LEA	B-41
	LINK	B-42
	LSL, LSR	B-43
	MOVE	B-45
	MOVE to CCR	B-46
	MOVE to SR	B-47
	MOVE from SR	B-48
	MOVE USP	B-49
	MOVEA	B-50
	MOVEM	B-51
	MOVEP	B-53
	MOVEQ	B-55
	MULS	B-56
	MULU	B-57
	NBCD	B-58
	NEG	B-59
	NEGX	B-60
	NOP	B-61
	NOT	B-62
	OR	B-63
	ORI	B-64
	PEA	B-65
	RESET	B-66
	ROL, ROR	B-67
	ROXL, ROXR	B-69
	RTE	B-71
	RTR	B-72
	RTS	B-73
	SBCD	B-74
	Scc	B-75
	STOP	B-76
	SUB	B-77
	SUBA	B-78
	SUBI	B-79
	SUBQ	B-80
	SUBX	B-81
	SWAP	B-83

Table of Contents (Continued)

Paragraph No.	Title	Page No.
	TAS	B-84
	TRAP	B-85
	TRAPV	B-86
	TST	B-87
	UNLK	B-88

Appendix C
Instruction Format Summary

C.0	Introduction	C-1
C.1	Bit Manipulation, Move Peripheral, Immediate Instructions	C-1
C.2	Move Byte Instruction	C-2
C.3	Move Long Instruction	C-2
C.4	Move Word Instruction	C-2
C.5	Miscellaneous Instructions	C-3
C.6	Add Quick, Subtract Quick, Set Conditionally, Decrement Instructions	C-5
C.7	Branch Conditionally Instruction	C-6
C.8	Move Quick Instruction	C-6
C.9	Or, Divide, Subtract Decimal Instructions	C-6
C.10	Subtract, Subtract Extended Instructions	C-7
C.11	Compare, Exclusive OR Instructions	C-7
C.12	Multiply, Add Decimal, Exchange Instructions	C-7
C.13	Add, Add Extended Instructions	C-8
C.14	Shift/Rotate Instructions	C-8

Appendix D
Instruction Execution Times

D.0	Introduction	D-1
D.1	Effective Address Operand Calculation Timing	D-1
D.2	Move Instruction Timing	D-2
D.3	Standard Instruction Timing	D-3
D.4	Immediate Instruction Timing	D-4
D.5	Single Operand Instruction Timing	D-4
D.6	Shift/Rotate Instruction Timing	D-5
D.7	Bit Manipulation Instruction Timing	D-5
D.8	Conditional Instruction Timing	D-6
D.9	JMP, JSR, LEA, PEA, MOVEM Instruction Timing	D-6
D.10	Multi-Precision Instructions Timing	D-7
D.11	Miscellaneous Instructions Timing	D-7
D.12	Exception Processing Timing	D-8

Table of Contents (Concluded)

Paragraph No.	*Title*	*Page No.*
	Appendix E **Pinout Diagram**	
E.0	Introduction	E-1
	Appendix F **Cross Macro Assembler**	
F.0	Introduction	F-1
F.1	M68000 Cross Macro Assembler	F-1
F.2	Instruction Set Summary	F-3

LIST OF ILLUSTRATIONS

Figure No.	Title	Page No.
1-1	HMOS Circuit Design Techniques	1-1
1-2	Programming Model	1-3
1-3	Status Register	1-4
1-4	Motorola's Microprocessor Evolution	1-10
2-1	Word Organization in Memory	2-2
2-2	Data Organization in Memory	2-2
2-3	Instruction Format	2-4
2-4	Single Effective Address Instruction Operation Word-General Format	2-5
4-1	Input and Output Signals	4-1
4-2	Word Read Cycle Flow Chart	4-8
4-3	Byte Read Cycle Flow Chart	4-9
4-4	Read and Write Cycle Timing Diagram	4-10
4-5	Word and Byte Read Cycle Timing Diagram	4-10
4-6	Word Write Cycle Flow Chart	4-12
4-7	Byte Write Cycle Flow Chart	4-13
4-8	Word and Byte Write Cycle Timing Diagram	4-14
4-9	Read-Modify-Write Cycle Flow Chart	4-16
4-10	Read-Modify-Write Cycle Timing Diagram	4-17
4-11	Bus Arbitration Cycle Flow Chart	4-19
4-12	Bus Arbitration Cycle Timing Diagram	4-20
4-13	Bus Error Timing Diagram	4-21
4-14	Re Run Bus Cycle Timing Diagram	4-22
4-15	Halt Signal Timing Characteristics	4-23
4-16	Simplified Single-Step Circuit	4-24
4-17	Reset Operation Timing Diagram	4-25
5-1	Exception Vector Format	5-4
5-2	Peripheral Vector Number Format	5-4
5-3	Address Translated from 8-Bit Vector Number	5-4
5-4	Interrupt Acknowledge Sequence Flow Chart	5-9
5-5	Interrupt Acknowledge Sequence Timing Diagram	5-10
5-6	Supervisor Stack Order for Bus Error Exception	5-13
6-1	M6800 Interfacing Flow Chart	6-1
6-2	MC68000 to M6800 Peripheral Timing Diagram	6-3
6-3	M6800 Interface — Example 1	6-4
6-4	M6800 Interface — Example 2	6-5
6-5	Autovector Operation Timing Diagram	6-6
E-1	Pinout Diagram	E-1
F-1	M68000 Cross Macro Assembler Operation	F-1

LIST OF TABLES

Table No.	Title	Page No.
1-1	Data Addressing Modes	1-5
1-2	Instruction Set	1-6
1-3	Variations of Instruction Types	1-7
2-1	Effective Address Encoding Summary	2-12
2-2	Implicit Instruction Reference Summary	2-13
3-1	Data Movement Operations	3-2
3-2	Integer Arithmetic Operations	3-3
3-3	Logical Operations	3-3
3-4	Shift and Rotate Operations	3-4
3-5	Bit Manipulation Operations	3-5
3-6	Binary Coded Decimal Operations	3-5
3-7	Program Control Operations	3-7
3-8	System Control Operations	3-7
4-1	Data Strobe Control of Data Bus	4-2
4-2	Function Code Outputs	4-5
4-3	Signal Summary	4-6
5-1	Reference Classification	5-3
5-2	Exception Vector Assignment	5-5
5-3	Exception Grouping and Priority	5-7
A-1	Condition Code Computations	A-3
A-2	Conditional Tests	A-4
B-1	Effective Addressing Mode Categories	B-1
C-1	Operation Code Map	C-1
D-1	Effective Address Operand Calculation Timing	D-1
D-2	Move Byte and Word Instruction Timing	D-2
D-3	Move Long Instruction Timing	D-2
D-4	Standard Instruction Timing	D-3
D-5	Immediate Instruction Timing	D-4
D-6	Single Operand Instruction Timing	D-4
D-7	Shift and Rotate Instruction Timing	D-5
D-8	Bit Manipulation Instruction Timing	D-5
D-9	Conditional Instruction Timing	D-6
D-10	JMP, JSR, LEA, PEA, MOVEM Instruction Timing	D-6
D-11	Multi-Precision Instructions Timing	D-7
D-12	Miscellaneous Instructions Timing	D-7
D-13	Exception Processing Timing	D-8
F-1	Instruction Set Summary	F-3

CHAPTER 1
GENERAL DESCRIPTION

1.0 INTRODUCTION

This section contains a general description of the MC68000 Microprocessor.

1.1 OVERVIEW

Advances in semiconductor technology have provided the capability to place on a single silicon chip a microprocessor at least an order of magnitude higher in performance and circuit complexity than has been previously available. The MC68000 is the first of a family of such VLSI microprocessors from Motorola. It combines state-of-the-art technology and advanced circuit design techniques with computer sciences to achieve an architecturally advanced 16-bit microprocessor. The high density of active elements coupled with an order of magnitude increase in performance over the original MC6800 is the direct result of significant advances in semiconductor technology. Advances such as dry plasma etching, projection printing, and HMOS (high-density, short-channel MOS) circuit design techniques shown in figure 1-1 have provided a sound technological base that has allowed Motorola's system engineers, computer scientists and marketing engineers a large degree of innovative freedom. The goals of applying this innovative freedom to microprocessors are to make the microprocessor easy to use, more reliable and more flexible for applications, while maximizing performance.

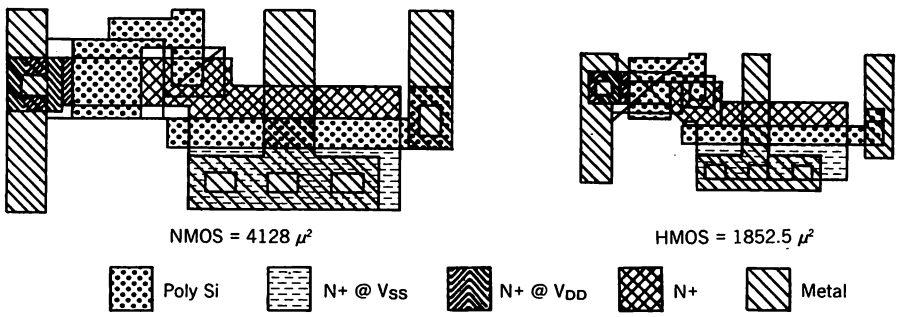


Figure 1-1. HMOS Circuit Design Techniques

The resources available to the MC68000 user consist of the following:

- 32-bit data and address registers
- 16 megabyte direct addressing range
- 56 powerful instruction types
- Operations on five main data types
- Memory mapped I/O
- 14 addressing modes

Particular emphasis has been given to the architecture to make it regular with respect to the registers, instructions (including all addressing modes), and data types. A consistent structure makes the architecture easy to learn and program, and, in the process, reduces both the time required to write programs and the space required to store programs. The net result is a great reduction in the cost and risk of developing software.

High systems throughput (up to an aggregate of two million instruction and data word transfers per second) is achieved even with readily available standard product memories with comparatively slow access times. The design flexibility of the data bus allows the mixing of slow and fast memories or peripherals with the processor, automatically optimizing the transfer rate on every access to keep the system operating at peak efficiency.

The hardware design of the CPU was heavily influenced by advances made in software technology. High-level language compilers as well as code produced from high-level languages must run efficiently on the new generation 16-bit and 32-bit microprocessors. The MC68000 supports high-level languages with its consistent architecture, multiple registers and stacks, large addressing range and high level language oriented instructions (LINK, UNLK, CHK, etc.). Also, operating systems for controlling the software operating environment of the MC68000 are supported by privileged instructions, memory management, a powerful vectored multi-level interrupt and trap structure, and specific instructions (MOVEP, MOVEM, TRAP, etc.).

The processor also provides both hardware and software interlocks for multiprocessor systems. The MC68000 contains bus arbitration logic for a shared bus and shared memory environment (shared with other MC68000 processors, DMA devices, etc.). Multiprocessor systems are also supported with software instructions (TAS — test and set operand). The MC68000 offers the maximum flexibility for microprocessor-based multiprocessor systems.

Advanced architecture processors must not only offer efficient solutions to large complex problems but must be able to handle the small, simple problems with proportional efficiency. The MC68000 has been designed to offer the maximum in performance and versatility to solve simple and complex problems efficiently.

As shown in figure 1-2, the MC68000 offers seventeen 32-bit registers in addition to the 32-bit program counter and 16-bit status register. The first eight registers (D0-D7)

are used as data registers for byte (8-bit), word (16-bit) and long word (32-bit) data operations. The second set of seven registers (A0-A6) and the system stack pointer (A7) may be used as software stack pointers and base address registers. In addition, these registers may be used for word and long word address operations. All of the seventeen registers may be used as index registers.

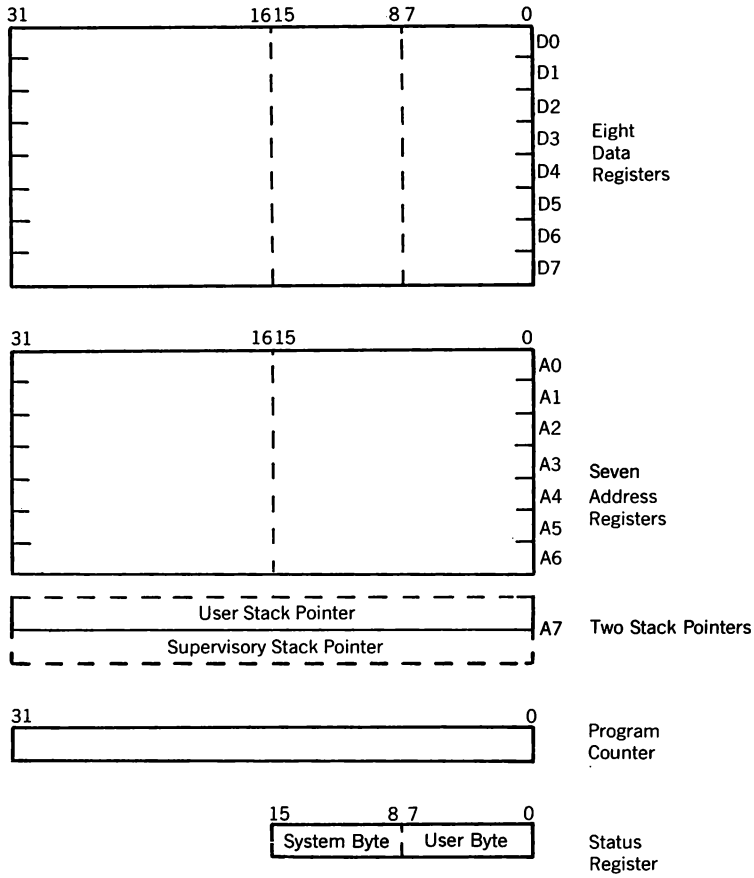


Figure 1-2. Programming Model

The 24-bit address bus provides a memory addressing range of more than 16 megabytes (actually 16,777,216 bytes). This large range of addressing capability, coupled with a memory management unit, allows large, modular programs to be developed and operated without resorting to cumbersome and time consuming software bookkeeping and paging techniques.

The status register, shown in figure 1-3, contains the interrupt mask (8 levels available) as well as the condition codes; overflow (V), zero (Z), negative (N), carry (C), and extend (X). Additional status bits indicate that the processor is in a trace (T) mode and/or in a supervisor (S) state. Ample space remains in the status register for future extensions of the MC68000 family.

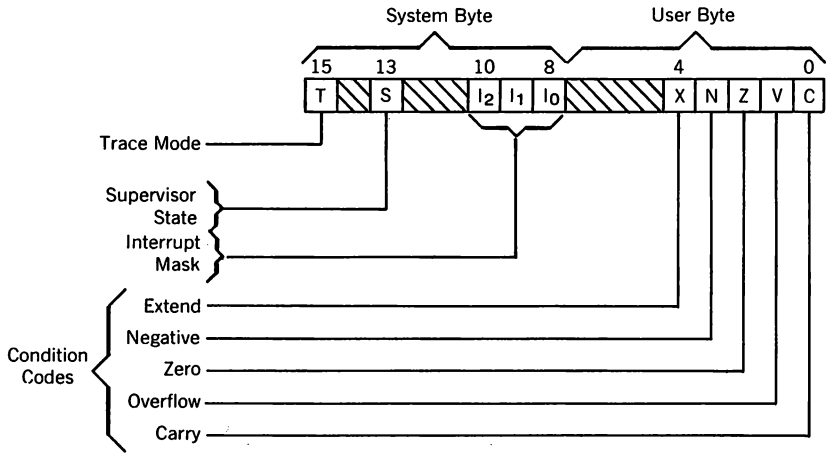


Figure 1-3. Status Register

Five basic data types are supported. These data types are:

- Bits
- BCD digits (4-bits)
- Bytes (8-bits)
- Words (16-bits)
- Long words (32-bits)

In addition, operations on other data types such as memory addresses, status word data, etc. are provided for in the instruction set.

The 14 flexible addressing modes, shown in table 1-1 include six basic types:

- Register Direct
- Register Indirect
- Absolute
- Immediate
- Program Counter Relative
- Implied

Included in the register indirect addressing modes is the capability to do post-incrementing, predecrementing, offsetting and indexing. Program counter relative mode can also be modified via indexing and offsetting.

Table 1-1. Data Addressing Modes

Mode	Generation
Register Direct Addressing	
Data Register Direct	EA = Dn
Address Register Direct	EA = An
Absolute Data Addressing	
Absolute Short	EA = (Next Word)
Absolute Long	EA = (Next Two Words)
Program Counter Relative Addressing	
Relative with Offset	EA = (PC) + d_{16}
Relative with Index and Offset	EA = (PC) + (Xn) + d_8
Register Indirect Addressing	
Register Indirect	EA = (An)
Postincrement Register Indirect	EA = (An), An ← An + N
Predecrement Register Indirect	An ← An - N, EA = (An)
Register Indirect With Offset	EA = (An) + d_{16}
Indexed Register Indirect With Offset	EA = (An) + (Xn) + d_8
Immediate Data Addressing	
Immediate	DATA = Next Word(s)
Quick Immediate	Inherent Data
Implied Addressing	
Implied Register	EA = SR, USP, SP, PC

NOTES:

EA = Effective Address
An = Address Register
Dn = Data Register
Xn = Address or Data Register used as Index Register
SR = Status Register
PC = Program Counter
d_8 = Eight-bit Offset (displacement)
d_{16} = Sixteen-bit Offset (displacement)
N = 1 for Byte, 2 for Words and 4 for Long Words
( ) = Contents of
← = Replaces

The MC68000 instruction set is shown in table 1-2. Some additional instructions are variations, or subsets of these and they appear in table 1-3. Special emphasis has been given to the instruction set's support of structured high-level languages to facilitate ease of programming. Each instruction, with few exceptions, operates on bytes, words, and long words and most instructions can use any of the 14 addressing modes. Combining instruction types, data types, and addressing modes, over 1000 useful instructions are provided. These instructions include signed and unsigned multiply and divide, "quick" arithmetic operations, BCD arithmetic and expanded operations (through traps). Additionally, its highly-symmetric, proprietary microcoded structure provides a sound, flexible base for the future.

Table 1-2. Instruction Set

Mnemonic	Description
ABCD	Add Decimal with Extend
ADD	Add
AND	Logical And
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
Bcc	Branch Conditionally
BCHG	Bit Test and Change
BCLR	Bit Test and Clear
BRA	Branch Always
BSET	Bit Test and Set
BSR	Branch to Subroutine
BTST	Bit Test
CHK	Check Register Against Bounds
CLR	Clear Operand
CMP	Compare
DBcc	Test Cond., Decrement and Branch
DIVS	Signed Divide
DIVU	Unsigned Divide
EOR	Exclusive Or
EXG	Exchange Registers
EXT	Sign Extend
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link Stack
LSL	Logical Shift Left
LSR	Logical Shift Right
MOVE	Move
MOVEM	Move Multiple Registers
MOVEP	Move Peripheral Data
MULS	Signed Multiply
MULU	Unsigned Multiply
NBCD	Negate Decimal with Extend
NEG	Negate
NOP	No Operation
NOT	One's Complement
OR	Logical Or
PEA	Push Effective Address
RESET	Reset External Devices
ROL	Rotate Left without Extend
ROR	Rotate Right without Extend
ROXL	Rotate Left with Extend
ROXR	Rotate Right with Extend
RTE	Return from Exception
RTR	Return and Restore
RTS	Return from Subroutine

Table 1-2. Instruction Set (continued)

Mnemonic	Description
SBCD	Subtract Decimal with Extend
S_{cc}	Set Conditional
STOP	Stop
SUB	Subtract
SWAP	Swap Data Register Halves
TAS	Test and Set Operand
TRAP	Trap
TRAPV	Trap on Overflow
TST	Test
UNLK	Unlink

Table 1-3. Variations of Instruction Types

Instruction Type	Variation	Description
ADD	**ADD**	Add
	ADDA	Add Address
	ADDQ	Add Quick
	ADDI	Add Immediate
	ADDX	Add with Extend
AND	**AND**	Logical And
	ANDI	And Immediate
CMP	**CMP**	Compare
	CMPA	Compare Address
	CMPM	Compare Memory
	CMPI	Compare Immediate
EOR	**EOR**	Exclusive Or
	EORI	Exclusive Or Immediate
MOVE	**MOVE**	Move
	MOVEA	Move Address
	MOVEQ	Move Quick
	MOVE from SR	Move from Status Register
	MOVE to SR	Move to Status Register
	MOVE to CCR	Move to Condition Codes
	MOVE to USP	Move to User Stack Pointer
NEG	**NEG**	Negate
	NEGX	Negate with Extend
OR	**OR**	Logical Or
	ORI	Or Immediate
SUB	**SUB**	Subtract
	SUBA	Subtract Address
	SUBI	Subtract Immediate
	SUBQ	Subtract Quick
	SUBX	Subtract with Extend

1.2 SOFTWARE DEVELOPMENT

Many innovative features have been incorporated to make programming easier, faster and more reliable.

1.2.1 CONSISTENT STRUCTURE. The highly regular structure of the MC68000 greatly simplifies the effort required to write programs in assembly language as well as high-level languages. Operations on integer data in registers and memory are independent of the data. Separate special instructions that operate on byte (8-bit), word (16-bit) and long word (32-bit) integers are not necessary. The programmer need only remember one mnemonic for each type of operation and then specify data size, source addressing mode and destination addressing mode. This has helped keep the total number of instructions small.

The dual operand nature of many of the instructions significantly increases the flexibility and power of the MC68000. Consistency is again maintained since all data registers and memory locations may be either a source or destination for most operations on integer data.

The addressing modes have been kept simple without sacrificing efficiency. All fourteen addressing modes operate consistently and are independent of the instruction operation itself. Additionally, all address registers may be used for the direct, register indirect and indexed addressing modes (immediate, program counter relative and absolute addressing by definition do not use address registers). For increased flexibility, any address or data register may be used as an index register. Address register consistency is maintained for stacking operations since any of the eight address registers may be utilized as user program stack pointers with the register indirect postincrement/predecrement addressing modes. Address register A7, however, is a special register that, in addition to its normal addressing capability, functions as the system stack pointer for stacking the program counter for subroutine calls as well as stacking the program counter and status register for traps and interrupts (while in the supervisor state).

1.2.2 STRUCTURED MODULAR PROGRAMMING. The art of programming microprocessors has evolved rapidly in the past few years. Numerous advanced techniques have been developed to allow easier, more consistent and reliable generation of software. In general, these techniques require that the programmer be more disciplined in observing a defined programming structure such as modular programming. Modular programming allows a required function or process to be broken down in short modules or subroutines that are concisely defined and easily programmed and tested. Such a technique is greatly simplified by the availability of advanced structured assemblers and block structured high-level languages such as PASCAL. Such concepts are virtually useless, however, unless parameters are easily transferred between and within software modules that operate on a reentrant and recursive basis. (To be reentrant a routine must be usable by interrupt and non-interrupt driven programs without the loss of data. A recursive routine is one that may call or use itself). The MC68000 provides the necessary architectural features to allow

efficient reentrant modular programming. The LINK and UNLK instructions reduce subroutine call overhead in two complementary instructions by allowing the manipulation of linked lists of data areas on the stack. The MOVEM (Move Multiple Registers) instruction also reduces subroutine call programming overhead. This allows moving, via an effective address, multiple registers that are specified by the programmer. Sixteen software trap vectors are provided with the TRAP instruction and are useful in operating system call routines or user generated macro routines. Other instructions that support modern structured programming techniques are PEA (Push Effective Address), LEA (Load Effective Address), RTR (Return and Restore), RTE (Return from Exception) as well as JSR, BSR and RTS.

The powerful vectored priority interrupt structure of the microprocessor allows straight-forward generation of reentrant modular input/output routines. Seven maskable levels of priority with 192 vector locations and seven autovector locations provide maximum flexibility for I/O control (a total of 255 vector locations are available for interrupts, hardware traps, and software traps).

1.2.3 IMPROVED SOFTWARE TESTABILITY. The MC68000 incorporates several features that reduce the chance for errors. Some of these features, such as consistent architecture and the structured modular programming capability, have already been discussed.

Of major importance to the systems programmer are features that have been incorporated specifically to detect the occurrence of programming errors or bugs. Several hardware traps, provided to indicate abnormal internal conditions, detect the following error conditions:

- Word access with an odd address
- Illegal instructions
- Unimplemented instructions
- Illegal memory access (bus error)
- Divide by zero
- Overflow condition code (separate instruction TRAPV)
- Register out of bounds (CHK instruction)
- Spurious Interrupt

Additionally, the sixteen software TRAP instructions may be utilized by the programmer to provide applications oriented error detection or correction routines.

An additional error detection tool is the CHK (Check Register against Bounds) instruction used for array bound checking by verifying that a data register contains a valid subscript. A trap occurs if the register contents are negative or greater than a limit.

Finally, the MC68000 includes a facility that allows instruction-by-instruction tracing of a program being debugged. This trace mode results in a trap being made to a

tracing routine after each instruction execution. The trace mode is available to the programmer when the microprocessor is in the supervisor state as well as the user state but may only be entered while in the supervisor state. The supervisor/user states provide an additional degree of error protection for the microprocessor by allowing memory protection of selected areas of memory when an external memory management device is used.

1.2.4 FUTURE FLEXIBILITY. Microprocessor VLSI circuit technology is advancing at an ever increasing rate. For example, the Motorola MC6800 — originally introduced in 1974 — has evolved into a number of more advanced products. This evolution has been along two paths, as shown in figure 1-4: increased functionality, with the MC6802 and MC6801 microcomputers, and increased performance with the MC68A00, MC68B00 and MC6809 microprocessors. The sound, well planned, architectural base provided by the original MC6800 made it possible to develop these improved products while taking full advantage of the major speed and density enhancements to NMOS VLSI. This was accomplished while maintaining an unprecedented degree of compatibility and consistency with the original MC6800 microprocessor.

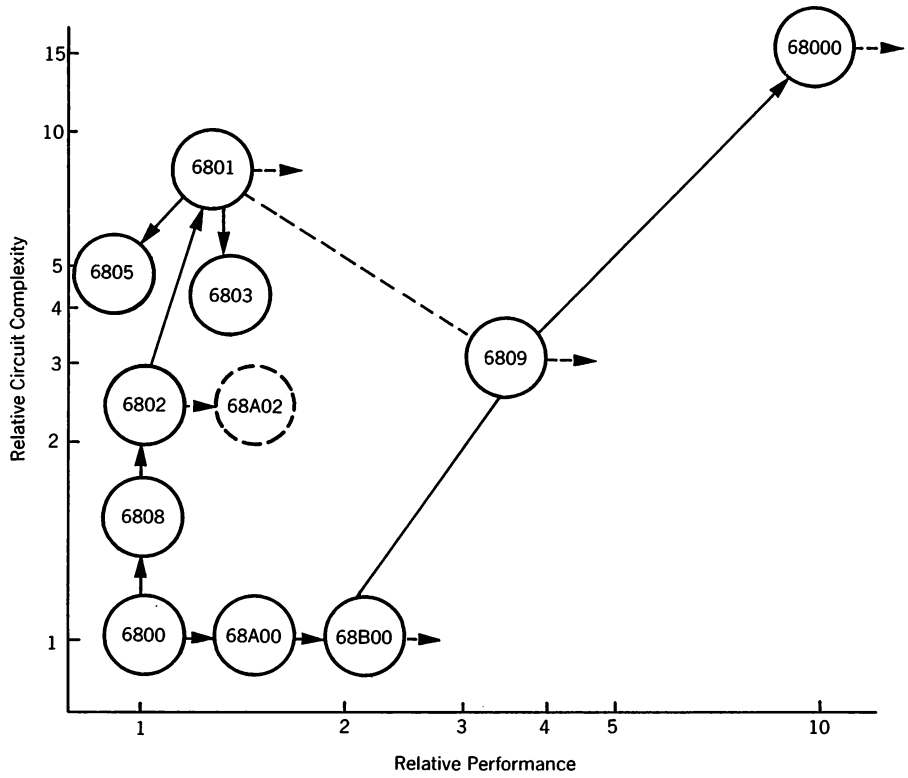


Figure 1-4. Motorola's Microprocessor Evolution

Similarly, a major consideration in the development of the MC68000 microprocessor has been to provide a good, solid, but flexible, base for future extendibility. Several architectural concepts have been incorporated that will allow this advanced product to be enhanced as semiconductor technological advances are made. For example, the highly consistent structure of the processor allows operations on 8-bit, 16-bit and 32-bit integers without the need for concatenation of registers or multiplexing of internal data buses. This regular structure of the processor lends itself to a more consistent, reliable design that can be easily expanded.

The MC68000 incorporates a proprietary multi-level microprogrammed structure that allows significant versatility in the implementation of instructions. In fact, more than one-eighth of the instruction op-code map has been set aside specifically for implementation of future instructions. In the interim, user implementation of instructions not currently in the instruction set is possible through the use of the trap instruction and the emulator traps.

1.2.5 MEMORY MANAGEMENT OF LARGE ADDRESSING SPACE. The ever decreasing costs of semiconductor memories in combination with the use of high-level languages and sophisticated disk operating systems allow Motorola's new generation of high performance microprocessors to be used in complex, memory intensive applications. In order to meet the needs of such applications, the MC68000 is capable of directly addressing more than 16 megabytes of memory. This large address space is directly accessed and managed very efficiently on a word or byte basis since operand size is specified by the instruction. The use of upper data strobe ($\overline{UDS}$) and lower data strobe ($\overline{LDS}$) signals allows easy access to high order bytes, low order bytes, or words.

Several additional useful features are provided that allow the programmer to efficiently manage memory usage. Powerful memory addressing modes such as register indirect, indexed, short and long absolute, and program counter relative allow well-ordered access to specific memory locations. These addressing modes allow easy address calculations (register indirect and indexed), direct access to memory location (short and long absolute) and position-independent or relocatable coding (program counter relative). Of course, the predecrement/postincrement register indirect addressing modes also allow efficient management of data in memory by permitting the programmer to generate as many as eight concurrent stacks or queues. Another feature that allows the programmer to manage the use of memory is the CHK (Check Register Against Bounds) instruction. This instruction permits the software implementation of a basic memory protection/management structure.

Still another significant feature provided in the MC68000 is the distinction between user and supervisor state. The supervisor state permits certain protected operations within the processor system. Of particular interest is that an external memory management unit may be used when the processor is in the user state to manage the large address space for the programmer. The memory management operations are transparent to the programmer when in the user state and can be changed or updated

only in the supervisor state. The memory management unit provides both management of a variable number of variable size segments (memory segmentation) and dynamic management of multi-task memory relocation and protection. The memory management unit regulates access to storage segments that are dedicated to read only data, read/write data, program code and protected data/code.

1.2.6 IMPROVED CODE DENSITY AND SPEED. With the advent of low cost, very high density VLSI RAMS and ROMS, it might incorrectly be assumed that the number of bytes of code needed to execute a given program is no longer important. Code density, however, is very critical, since microprocessor speed is highly dependent upon the number of executed instruction words. During the early development of the MC68000, extensive studies were made of the use of instructions and sequences of instructions in many microprocessor applications. These studies identified not only statically frequent instructions but also dynamically frequent instructions. (The dynamic frequency of instructions is a measure of how often an instruction is executed while static frequency is a measure of how often it occurs in a program listing or is encountered by an assembler.) The result of these studies is that the major contributer to the increased efficiency is the highly regular structure of the architecture. The consistency of the architecture, instruction set, and addressing modes significantly reduces the number of instructions needed to accomplish a given task. Additionally, many instructions have been included to specifically improve code density and speed. For example, single word add and subtract instructions using quick immediate addressing allow fast, small value arithmetic operations on data registers and memory. A move quick (MOVEQ) provides the ability to load a small (8-bit) signed word into any register in a single word operation. In order to improve the speed of loop operations, a single instruction for test condition, decrement and branch (DB_{cc}) is included. Of course, the TRAP, move multiple registers (MOVEM), link stack (LINK), unlink stack (UNLK) and check limit (CHK) instructions significantly reduce code requirements for subroutines, operating system calls and stacking operations.

Other instructions that help reduce coding requirements and improve performance of arithmetic operations are signed and unsigned multiply (MULS and MULU), signed and unsigned divide (DIVS and DIVU), BCD arithmetic (ABCD, SBCD), as well as the standard binary integer operations. In order to improve the efficiency of moving or transferring data, a powerful MOVE data instruction has been incorporated that allows the transfer of bytes, words and long words and operates in all data addressing modes. Thus, register-to-register, register-to-memory, memory-to-register, and memory-to-memory transfers are permitted.

In addition to the powerful instructions that provide a substantial improvement in processor throughput, numerous architectural features significantly reduce the execution times for all instructions. The separate (non-multiplexed) address and data buses, instruction pre-fetch pipeline and 32-bit internal registers are major contributors to the processor's unequaled performance.

1.2.7 SOFTWARE SUPPORT AND MC6800 COMPATIBILITY. The system designers and programmers using the MC68000 in an application have available a complete, compatible system of hardware and software. The microprocessor is supported by a full range of software development tools including disk operating systems, debug aids, assemblers, and high-level languages.

The careful planning of this new microprocessor provides a superset of the MC6800 instruction set enhanced by the addition of more and larger registers, powerful consistent structure and many flexible addressing modes. This allows efficient translation of existing MC6800 programs, which can then be further optimized by taking full advantage of the versatile and powerful features of the MC68000.

This careful planning of similarities between the MC68000 and the MC6800 does not stop at software compatibility (by translation) but also extends to peripheral controller interfacing. Motorola's extensive line of intelligent M6800 family peripherals (including MC6854 Advanced Data Link Controller and the MC68488 General Purpose Interface Adapter) can be directly and easily interfaced to the MC68000. Three signal lines: enable (E), valid memory address ($\overline{\text{VMA}}$) and valid peripheral address ($\overline{\text{VPA}}$) are provided to simplify the interface to Motorola's standard MC6800 peripherals.

CHAPTER 2
DATA ORGANIZATION AND ADDRESSING CAPABILITIES

2.0 INTRODUCTION

This section describes the data organization and addressing capabilities of the MC68000.

2.1 OPERAND SIZE

Operand sizes are defined as follows: a byte equals 8 bits, a word equals 16 bits, and a long word equals 32 bits. The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation. All explicit instructions support byte, word or long operands. Implicit instructions support some subset of all three sizes.

2.2 DATA ORGANIZATION IN REGISTERS

The eight data registers support data operands of 1, 8, 16, or 32 bits. The seven address registers together with the active stack pointer support address operands of 32 bits.

2.2.1 DATA REGISTERS. Each data register is 32 bits wide. Byte operands occupy the low order 8 bits, word operands the low order 16 bits, and long operands the entire 32 bits. The least significant bit is addressed as bit zero; the most significant bit is addressed as bit 31.

When a data register is used as either a source or destination operand, only the appropriate low order portion is changed; the remaining high-order portion is neither used nor changed.

2.2.2 ADDRESS REGISTERS. Each address register and the stack pointer is 32 bits wide and holds a full 32 bit address. Address registers do not support byte sized operands. Therefore, when an address register is used as a source operand, either the low order word or the entire long operand is used depending upon the operation size. When an address register is used as the destination operand, the entire register is

affected regardless of the operation size. If the operation size is word, any other operands are sign extended to 32 bits before the operation is performed.

2.3 DATA ORGANIZATION IN MEMORY

Bytes are individually addressable with the high order byte having an even address the same as the word as shown in figure 2-1. The low order byte has an odd address that is one count higher than the word address. Instructions and multibyte data are accessed only on word (even byte) boundaries. If a long word datum is located at address n (n even), then the second word of that datum is located at address n + 2.

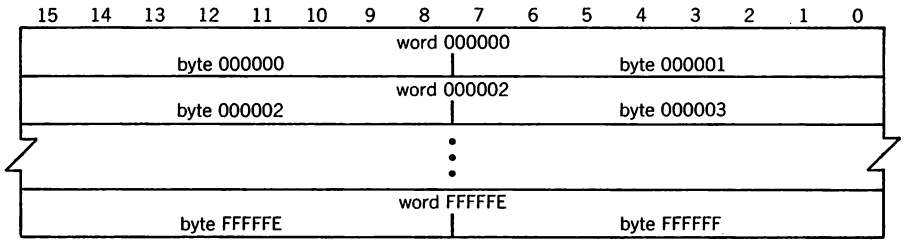


Figure 2-1. Word Organization In Memory

The data types supported by the MC68000 are: bit data, integer data of 8, 16, and 32 bits, 32-bit addresses and binary coded decimal data. Each of these data types is put in memory as shown in figure 2-2.

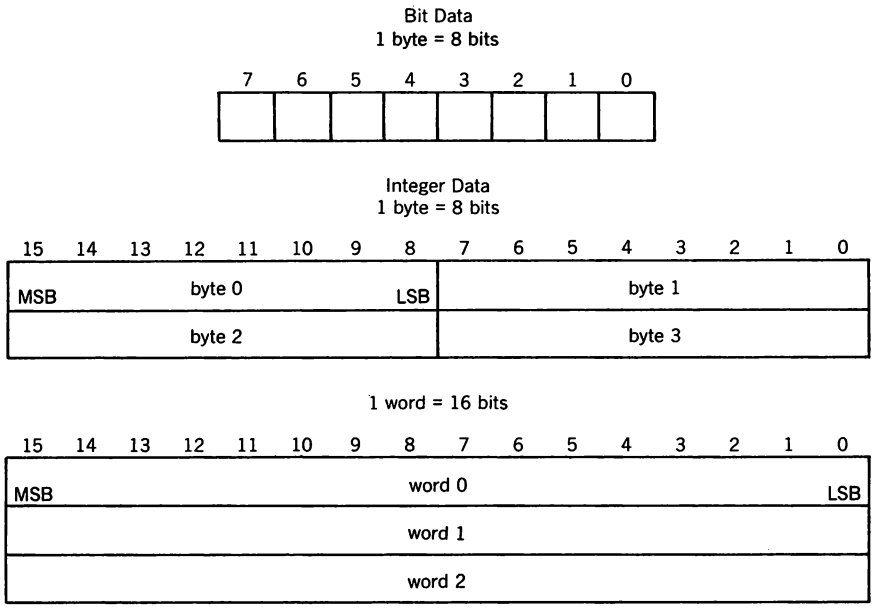

Bit Data
1 byte = 8 bits
7 6 5 4 3 2 1 0
Integer Data
1 byte = 8 bits
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
MSB
byte 0
LSB
byte 1
byte 2
byte 3
1 word = 16 bits
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
MSB
word 0
LSB
word 1
word 2

Figure 2-2. Data Organization In Memory (Sheet 1 Of 2)

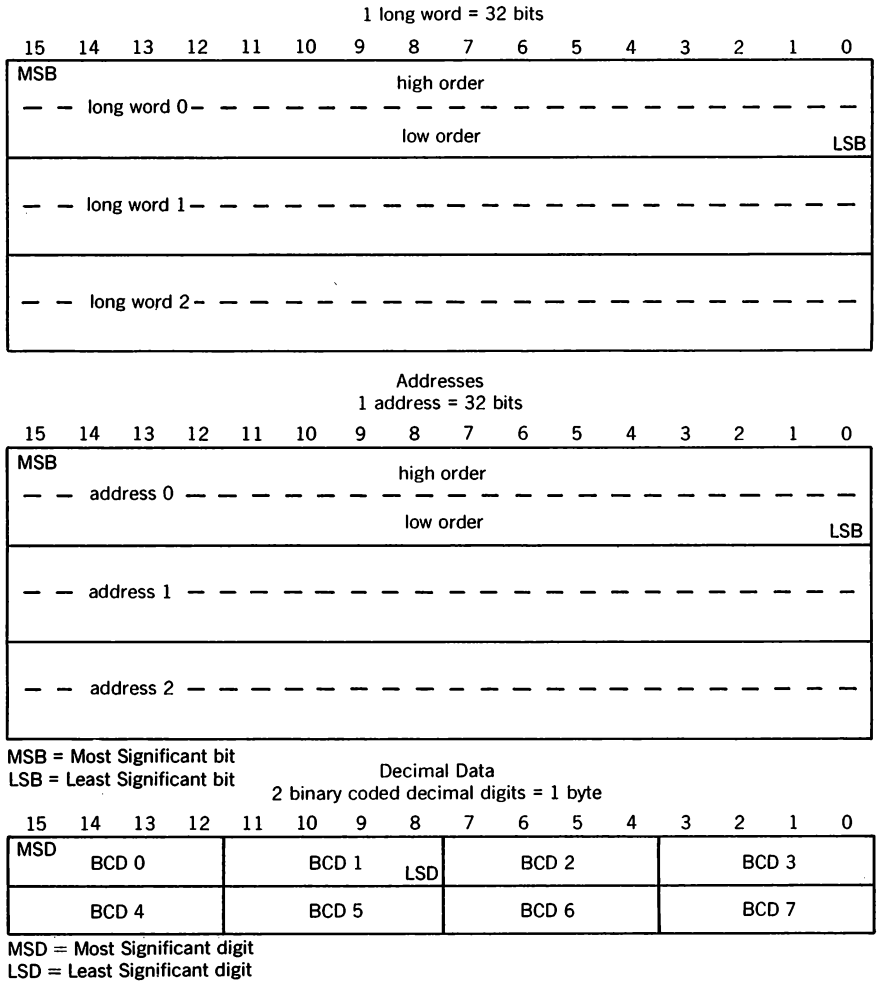


Figure 2-2. Data Organization In Memory (Sheet 2 Of 2)

2.4 ADDRESSING

Instructions for the MC68000 contain two kinds of information: the type of function to be performed, and the location of the operand(s) on which to perform that function. The methods used to locate (address) the operand(s) are explained in the following paragraphs.

Instructions specify operand location in one of three ways:

Register Specification — the number of the register is given in the register field of the instruction.

Effective Address — use of the different effective address modes.

Implicit Reference — the definition of certain instructions implies the use of specific registers.

2.5 INSTRUCTION FORMAT

Instructions are from one to five words in length as shown in figure 2-3. The length of the instruction and the operation to be performed is specified by the first word of the instruction which is called the operation word. The remaining words further specify the operands. These words are either immediate operands or extensions to the effective address mode specified in the operation word.

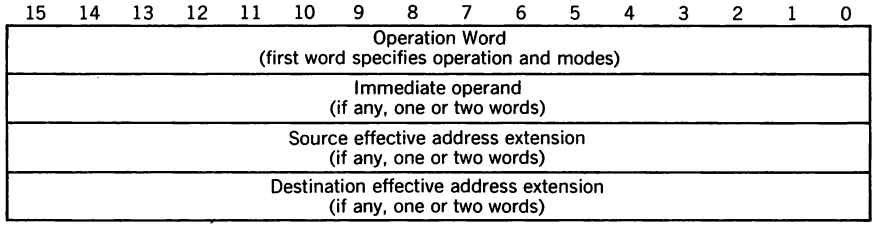

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Operation Word
(first word specifies operation and modes)
Immediate operand
(if any, one or two words)
Source effective address extension
(if any, one or two words)
Destination effective address extension
(if any, one or two words)

Figure 2-3. Instruction Format

2.6 PROGRAM/DATA REFERENCES

The MC68000 separates memory references into two classes: program references, and data references. Program references, as the name implies, are references to that section of memory that contains the program being executed. Data references refer to that section of memory that contains data. Generally, operand reads are from the data space. All operand writes are to the data space.

2.7 REGISTER NOTATION

Appendix B contains a definition of the register transfer language (RTL) used in describing instruction operations. The RTL description of registers identifies the registers as follows:

An — address register (n specifies the register number)
Dn — data register (n specifies the register number)
Rn — any register, address or data (n specifies the register number)
PC — program counter
SR — status register
CCR — condition code half of the status register
SP — the active stack pointer (either user or supervisor)
USP — user stack pointer
SSP — supervisor stack pointer
d — displacement value
N — operand size in bytes (1, 2, 4)

2.8 ADDRESS REGISTER INDIRECT NOTATION

When an address register is used to point to a memory location, the addressing mode is called address register indirect. The term indirect is used because the operation of the instruction is not directed to the address register itself, but to the memory location pointed to by the address register. The RTL symbol for the indirect mode is an address

register designation followed by an at symbol (@). The notation A4@ indicates that the content of address register four points to the memory location that will be used as the operand.

2.9 REGISTER SPECIFICATION

The register field within an instruction specifies the register to be used. Other fields within the instruction specify whether the register selected is an address or data register and how the register is to be used.

2.10 EFFECTIVE ADDRESS

Most instructions specify the location of an operand by using the effective address field in the operation word. For example, figure 2-4 shows the general format of the single effective address instruction operation word. The effective address is composed of two 3-bit fields: the mode field, and the register field. The value in the mode field selects the different address modes. The register field contains the number of a register.

The effective address field may require additional information to fully specify the operand. This additional information, called the effective address extension, is contained in a following word or words and is considered part of the instruction as shown in figure 2-3. The effective address modes are grouped into three categories: register direct, memory addressing, and special.

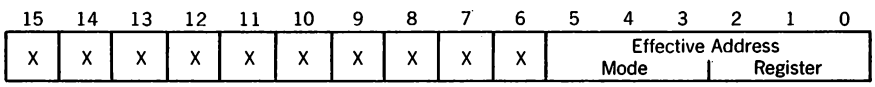


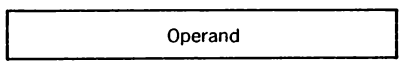
Figure 2-4. Single Effective Address Instruction Operation Word-General Format

2.10.1 REGISTER DIRECT MODES. These effective addressing modes specify that the operand is in one of the 16 multifunction registers.

2.10.1.1 Data Register Direct. The operand is in the data register specified by the effective address register field.

Generation: EA = Dn
RTL Notation: Dn
Mode: 000
Register: n

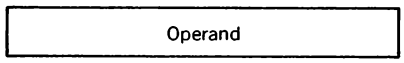
Data Register Dn



2.10.1.2 Address Register Direct. The operand is in the address register specified by the effective address register field.

Generation: EA = An
RTL Notation: An
Mode: 001
Register: n

Address Register An



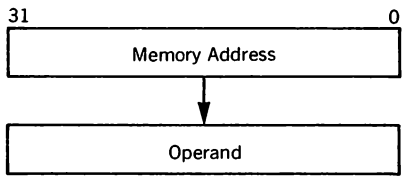
2.10.2 MEMORY ADDRESS MODES. These effective addressing modes specify that the operand is in memory and provide the specific address of the operand.

2.10.2.1 Address Register Indirect. The address of the operand is in the address register specified by the register field. The reference is classified as a data reference with the exception of the jump and jump to subroutine instructions.

Generation: EA = (An)
RTL Notation: An@
Mode: 010
Register: n

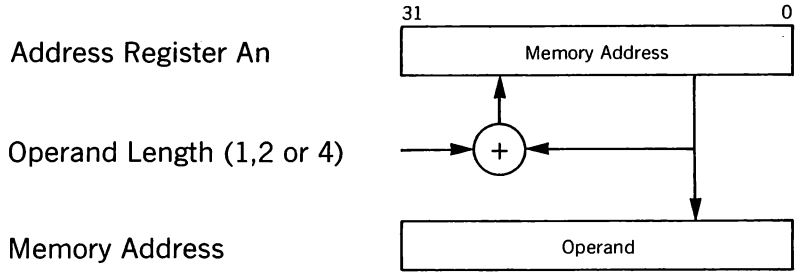
Address Register An

Memory Address



2.10.2.2 Address Register Indirect With Postincrement. The address of the operand is in the address register specified by the register field. After the operand address is used, it is incremented by one, two, or four depending upon whether the size of the operand is byte, word, or long. If the address register is the stack pointer and the operand size is byte, the address is incremented by two rather than one to keep the stack pointer on a word boundary. The reference is classified as a data reference.

Generation: EA = (An)
An = An + N
RTL Notation: An@+
Mode: 011
Register: n



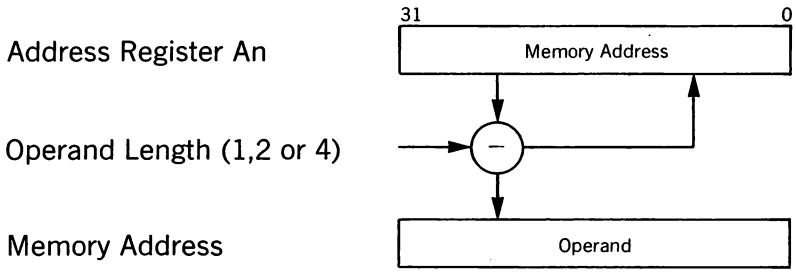
2.10.2.3 Address Register Indirect With Predecrement. The address of the operand is in the address register specified by the register field. Before the operand address is used, it is decremented by one, two, or four depending upon whether the operand size is byte, word, or long. If the address register is the stack pointer and the operand size is byte, the address is decremented by two rather than one to keep the stack pointer on a word boundary. The reference is classified as a data reference.

Generation: An = An - N
EA = (An)

RTL Notation: An@-

Mode: 100

Register: n



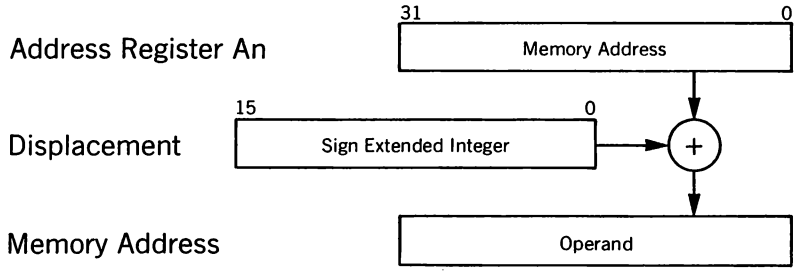
2.10.2.4 Address Register Indirect With Displacement. This address mode requires one word of extension. The address of the operand is the sum of the address in the address register and the sign-extended 16-bit displacement integer in the extension word. The reference is classified as a data reference with the exception of the jump and jump to subroutine instructions.

Generation: EA = (An) + d

RTL Notation: An@(d)

Mode: 101

Register: n



2.10.2.5 Address Register Indirect With Index. This address mode requires one word of extension formatted as shown below.

15	14 13 12	11	10	9	8	7 6 5 4 3 2 1 0
D/A	Register	W/L	0	0	0	Displacement Integer

Bit 15 —
Index register indicator
0 — data register
1 — address register

Bits 14 through 12 —
Index register number

Bit 11 —
Index Size
0 — sign-extended, low order integer in index register
1 — long value in index register

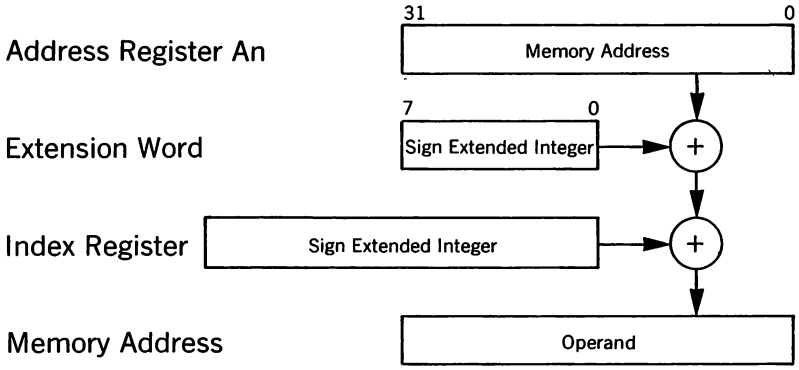
The address of the operand is the sum of the address in the address register, the sign-extended displacement integer in the low order eight bits of the extension word, and the contents of the index register. The reference is classified as a data reference with the exception of the jump and jump to subroutine instructions.

Generation: EA = (An) + (Ri) + d

RTL Notation: An@(d, Ri.W)
An@(d, Ri.L)

Mode: 110

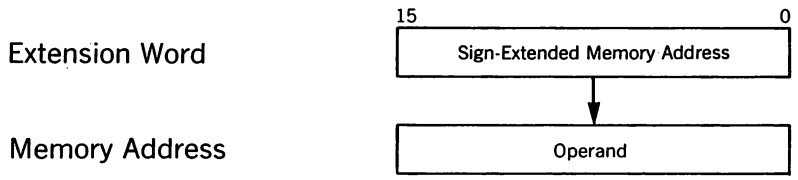
Register: n



2.10.3 SPECIAL ADDRESS MODES. The special address modes use the effective address register field to specify the special addressing mode instead of a register number.

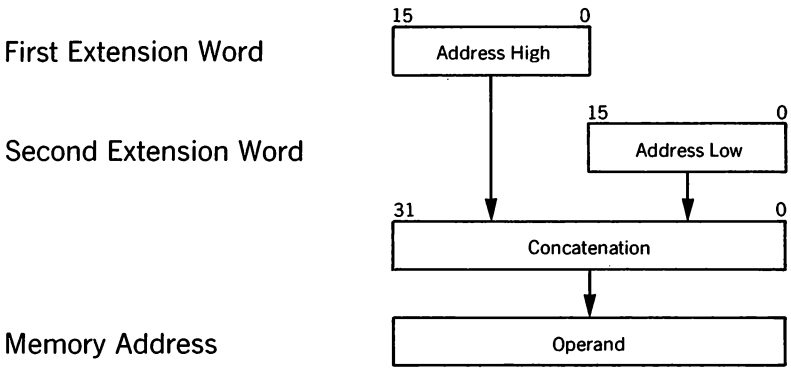
2.10.3.1 Absolute Short Address. This address mode requires one word of extension. The address of the operand is in the extension word. The 16-bit address is sign extended before it is used. The reference is classified as a data reference with the exception of the jump and jump to subroutine instructions.

Generation: EA given
RTL Notation: xxx.W
Mode: 111
Register: 000



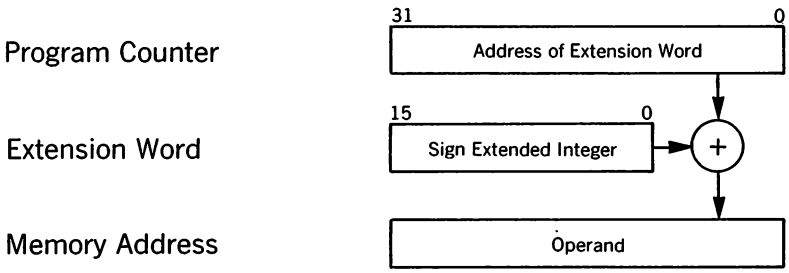
2.10.3.2 Absolute Long Address. This address requires two words of extension. The address of the operand is developed by the concatenation of the extension words. The high-order part of the address is the first extension word; the low order part of the address is the second extension word. The reference is classified as a data reference with the exception of the jump and jump to subroutine instructions.

Generation: EA given
RTL Notation: xxx.L
Mode: 111
Register: 001

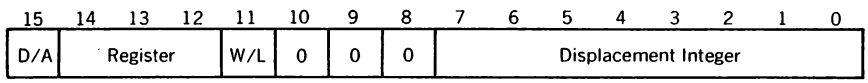


2.10.3.3 Program Counter With Displacement. This address mode requires one word of extension. The address of the operand is the sum of the address in the program counter and the sign-extended 16-bit displacement integer in the extension word. The value in the program counter is the address of the extension word. The reference is classified as a program reference.

Generation: EA = (PC) + d
RTL Notation: PC@(d)
Mode: 111
Register: 010



2.10.3.4 Program Counter With Index. This address mode requires one word of extension formatted as shown below.



Bit 15 —

Index register indicator
0 — data register
1 — address register

Bits 14 through 12 —

Index register number

Bit 11 —

Index size
0 — sign-extended, low order word integer in index register
1 — long value in index register

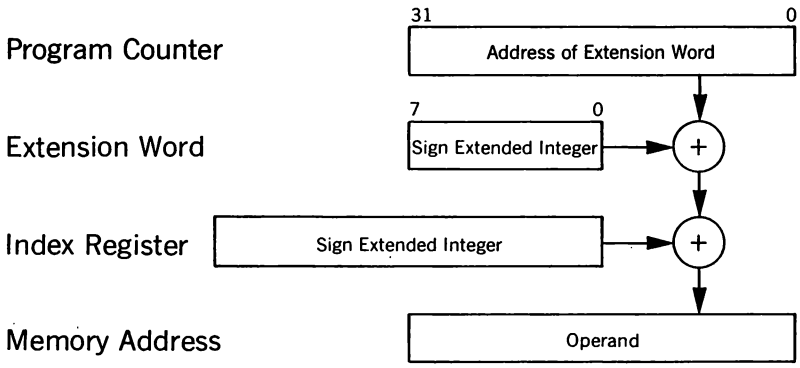
The address is the sum of the address in the program counter, the sign-extended displacement integer in the lower eight bits of the extension word, and the contents of the index register. The value in the program counter is the address of the extension word. This reference is classified as a program reference.

Generation: EA = (PC) + (Ri) + d

RTL Notation: PC@(d, Ri.W)
PC@(d, Ri.L)

Mode: 111

Register: 011



2.10.3.5 Immediate Data. This address mode requires either one or two words of extension depending on the size of the operation.

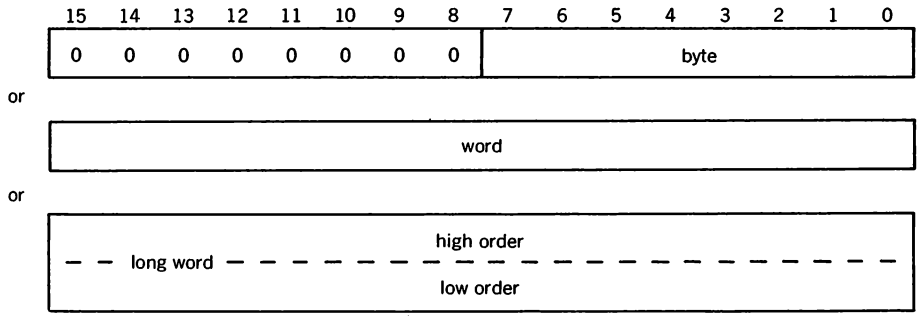
Byte operation — operand is low order byte of extension word

Word operation — operand is extension word

Long operation — operand is in the two extension words, high order 16-bits are in the first extension word, low order 16 bits are in the second extension word.

Generation: Operand given
RTL Notation: #xxxx
Mode: 111
Register: 100

The extension word formats are shown below:



2.10.3.6 Condition Codes Or Status Register. A selected set of instructions may reference the status register by means of the effective address field. These are:

ANDI to CCR EORI to CCR ORI to CCR
ANDI to SR EORI to SR ORI to SR

RTL Notation: CCR or SR
Mode: 111
Register: 100

2.10.4 EFFECTIVE ADDRESS ENCODING SUMMARY. Table 2-1 is a summary of the effective addressing modes discussed in the previous paragraphs.

Table 2-1. Effective Address Encoding Summary

Addressing Mode	Mode	Register
Data Register Direct	000	register number
Address Register Direct	001	register number
Address Register Indirect	010	register number
Address Register Indirect with Postincrement	011	register number
Address Register Indirect with Predecrement	100	register number
Address Register Indirect with Displacement	101	register number
Address Register Indirect with Index	110	register number
Absolute Short	111	000
Absolute Long	111	001
Program Counter with Displacement	111	010
Program Counter with Index	111	011
Immediate or Status Register	111	100

2.11 IMPLICIT REFERENCE

Some instructions make implicit reference to the program counter (PC), the system stack pointer (SSP), the user stack pointer (USP), or the status register (SR). Table 2-2 provides a list of these instructions and the registers implied.

Table 2-2. Implicit Instruction Reference Summary

Instruction	Implied Register(s)
Branch Conditional (B_{cc}), Branch Always (BRA)	PC
Branch to Subroutine (BSR)	PC, SP
Check Register against Bounds (CHK)	SSP, SR
Test Condition , Decrement and Branch (DB_{cc})	PC
Signed Divide (DIVS)	SSP, SR
Unsigned Divide (DIVU)	SSP, SR
Jump (JMP)	PC
Jump to Subroutine (JSR)	PC, SP
Link and Allocate (LINK)	SP
Move Condition Codes (MOVE CCR)	SR
Move Status Register (MOVE SR)	SR
Move User Stack Pointer (MOVE USP)	USP
Push Effective Address (PEA)	SP
Return from Exception (RTE)	PC, SP, SR
Return and Restore Condition Codes (RTR)	PC, SP, SR
Return from Subroutine (RTS)	PC, SP
Trap (TRAP)	SSP, SR
Trap on Overflow (TRAPV)	SSP, SR
Unlink (UNLK)	SP

2.12 STACKS AND QUEUES

In addition to supporting the array data structure with the index addressing mode, the MC68000 also supports stack and queue data structures with the address register indirect postincrement and predecrement addressing modes. A stack is a last-in-first-out (LIFO) list, a queue is a first-in-first-out (FIFO) list. When data is added to a stack or queue, it is "pushed" onto the structure; when it is removed, it is "pulled" from the structure.

The system stack is used implicitly by many instructions; user stacks and queues may be created and maintained through the addressing modes.

2.12.1 SYSTEM STACK. Address register seven (A7) is the system stack pointer (SP). The system stack pointer is either the supervisor stack pointer (SP) or the user stack pointer (USP), depending on the state of the S-bit in the status register. If the S-bit indicates supervisor state the SSP is the active system stack pointer, and the USP

cannot be referenced as an address register. If the S-bit indicates user state, the USP is the active system stack pointer, and the SSP cannot be referenced. Each system stack fills from high memory to low memory. The address mode SP@- creates a new item on the active system stack, and the address mode SP@+ deletes an item from the active system stack.

The program counter is saved on the active system stack on subroutine calls, and restored from the active system stack on returns. On the other hand, both the program counter and the status register are saved on the supervisor stack during the processing of traps and interrupts. Thus, the correct execution of the supervisor state code is not dependent on the behavior of user code and user programs may use the user stack pointer arbitrarily.

In order to keep data on the system stack aligned properly, data entry on the stack is restricted so that data is always put in the stack on a word boundary. Thus byte data is pushed on or pulled from the system stack in the high order half of the word; the lower half is unchanged.

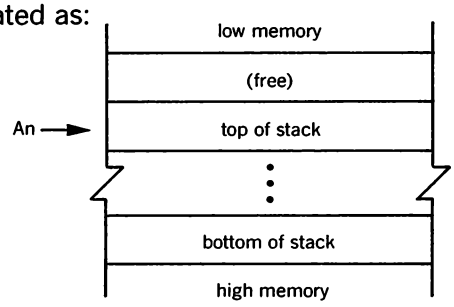
2.12.2 USER STACKS. User stacks can be implemented and manipulated by employing the address register indirect with postincrement and predecrement addressing modes. Using an address register (one of A0 through A6), the user may implement stacks which are filled either from high memory to low memory, or vice versa. The important things to remember are:

- using predecrement, the register is decremented before its contents are used as the pointer into the stack,
- using postincrement, the register is incremented after its contents are used as the pointer into the stack,
- byte data must be put on the stack in pairs when mixed with word or long data so that the stack will not get misaligned when the data is retrieved. Word and long accesses must be on word boundary (even) addresses.

Stack growth from high to low memory is implemented with

An@- to push data on the stack,
An@+ to pull data from the stack.

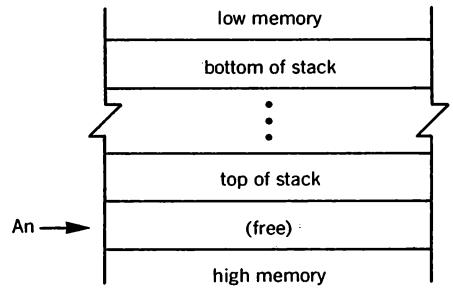
After either a push or a pull operation, register An points to the last (top) item on the stack. This is illustrated as:



Stack growth from low to high memory is implemented with

An@+ to push data on the stack,
An@- to pull data from the stack.

After either a push or a pull operation, register An points to the next available space on the stack. This is illustrated as:

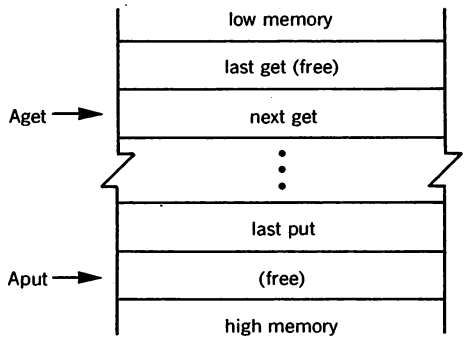


2.12.3 QUEUES. User queues can be implemented and manipulated with the address register indirect with postincrement or predecrement addressing modes. Using a pair of address registers (two of A0 through A6), the user may implement queues which are filled either from high memory to low memory, or vice versa. Because queues are pushed from one end and pulled from the other, two registers are used: the put and get pointers.

Queue growth from low to high memory is implemented with

Aput@+ to put data into the queue,
Aget@+ to get data from the queue.

After a put operation, the put address register points to the next available space in the queue and the unchanged get address register points to the next item to remove from the queue. After a get operation, the get address register points to the next item to remove from the queue and the unchanged put address register points to the next available space in the queue. This is illustrated as:

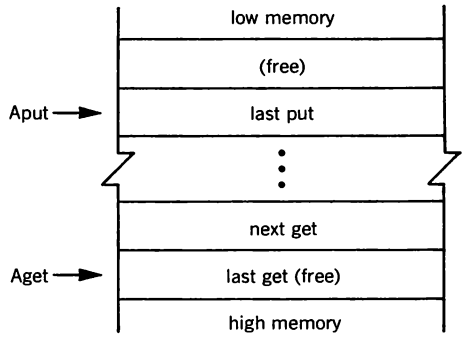


If the queue is to be implemented as a circular buffer, the address register should be checked and, if necessary, adjusted before the put or get operation is performed. The address register is adjusted by subtracting the buffer length (in bytes).

Queue growth from high to low memory is implemented with

Aput@- to put data into the queue,
Aget@- to get data from the queue.

After a put operation, the put address register points to the last item put in the queue, and the unchanged get address register points to the last item removed from the queue. After a get operation, the get address register points to the last item removed from the queue and the unchanged put address register points to the last item put in the queue. This is illustrated as:



If the queue is to be implemented as a circular buffer, the get or put operation should be performed first, and then the address register should be checked and, if necessary, adjusted. The address register is adjusted by adding the buffer length (in bytes).

CHAPTER 3
INSTRUCTION SET SUMMARY

3.0 INTRODUCTION

This chapter contains an overview of the form and structure of the MC68000 instruction set. The instructions form a set of tools that include all the machine functions to perform the following operations:

Data Movement
Integer Arithmetic
Shifts and Rotates
Bit Manipulation
Binary Coded Decimal
Program Control
System Control

The complete range of instruction capabilities combined with the flexible addressing modes described in Chapter 2 provide a very flexible base for program development. Detailed information about each instruction is given in Appendix B.

3.1 DATA MOVEMENT OPERATIONS

The basic method of data acquisition (transfer and storage) is provided by the move (MOVE) instruction. The move instruction and the effective addressing modes allow both address and data manipulation. Data move instructions allow byte, word, and long operands to be transferred from memory to memory, memory to register, register to memory, and register to register. Address move instructions allow word and long operand transfers and ensure that only legal address manipulations are executed. In addition to the general move instruction there are several special data movement instructions: move multiple registers (MOVEM), move peripheral data (MOVEP), exchange registers (EXG), load effective address (LEA), push effective address (PEA), link stack (LINK), unlink stack (UNLK), and move quick (MOVEQ). Table 3-1 is a summary of the data movement operations.

Table 3-1. Data Movement Operations

Instruction	Operand Size	Operation
EXG	32	Rx → Ry
LEA	32	EA → An
LINK	—	An → SP@- SP → An SP + d → SP
MOVE	8, 16, 32	(EA)s → EAd
MOVEM	16, 32	(EA) → An, Dn An, Dn → EA
MOVEP	16, 32	(EA) → Dn Dn → EA
MOVEQ	8	#Imm → Dn
PEA	32	EA → SP@-
SWAP	32	Dn [31:16] ↔ Dn [15:0]
UNLK	—	An → SP SP @+ → An

Notes:
s = source
d = destination
[ ] = bit numbers

3.2 INTEGER ARITHMETIC OPERATIONS

The arithmetic operations include the four basic operations of add (ADD), subtract (SUB), multiply (MUL), and divide (DIV) as well as arithmetic compare (CMP), clear (CLR), and negate (NEG). The add and subtract instructions are available for both address and data operations, with data operations accepting all operand sizes. Address operations are limited to legal address size operands (16 or 32 bits). Data, address, and memory compare operations are also available. The clear and negate instructions may be used on all sizes of data operands.

The multiply and divide operations are available for signed and unsigned operands using word multiply to produce a long product, and a long word dividend with word divisor to produce a word quotient with a word remainder.

Multiprecision and mixed size arithmetic can be accomplished using a set of extended instructions. These instructions are: add extended (ADDX), subtract extended (SUBX), sign extend (EXT), and negate binary with extend (NEGX).

A test operand (TST) instruction that will set the condition codes as a result of a compare of the operand with zero is also available. Test and set (TAS) is a synchronization useful in multiprocessor systems. Table 3-2 is a summary of the integer arithmetic operations.

Table 3-2. Integer Arithmetic Operations

Instruction	Operand Size	Operation
ADD	8, 16, 32	Dn + (EA) → Dn
		(EA) + Dn → EA
		(EA) + #xxx → EA
	16, 32	An + (EA) → An
ADDX	8, 16, 32	Dx + D6 + X → Dx
	16, 32	Ax@- Ay@- + X → Ax@
CLR	8, 16, 32	0 → EA
CMP	8, 16, 32	Dn - (EA)
		(EA) - #xxx
		Ax@+ - Ay@+
	16, 32	An - (EA)
DIVS	32 ÷ 16	Dn/(EA) → Dn
DIVU	32 ÷ 16	Dn/(EA) → Dn
EXT	8 → 16	$(Dn)_8 \rightarrow Dn_{16}$
	16 → 32	$(Dn)_{16} \rightarrow Dn_{32}$
MULS	16 * 16 → 32	Dn * (EA) → Dn
MULU	16 * 16 → 32	Dn * (EA) → Dn
NEG	8, 16, 32	0 - (EA) → EA
NEGX	8, 16, 32	0 - (EA) - X → EA
SUB	8, 16, 32	Dn - (EA) → Dn
		(EA) - Dn → EA
		(EA) - #xxx → EA
	16, 32	An - (EA) → An
SUBX	8, 16, 32	Dx - Dy - X → Dx
		Ax@- - Ay@- - X → Ax@
TAS	8	(EA) - 0, 1 → EA[7]
TST	8, 16, 32	(EA) - 0

Note:
[ ] = bit number

3.3 LOGICAL OPERATIONS

Logical operation instructions AND, OR, EOR, and NOT are available for all sizes of integer data operands. A similar set of immediate instructions (ANDI, ORI, and EORI) provide these logical operations with all sizes of immediate data. These immediate instructions also allow manipulation of the status register. Table 3-3 is a summary of the logical operations.

3.4 SHIFT AND ROTATE OPERATIONS

Shift operations in both directions are provided in the arithmetic instructions ASR and ASL and logical shift instructions LSR and LSL. The rotate instructions (with and without extend) available are ROXR, ROXL, ROR, and ROL. All shift and rotate operations can be performed in either registers or memory. Register shfits and rotates support all operand sizes and allow a shift count specified in the instruction of one to eight bits, or 0 to 63 bits specified in a data register.

Memory shifts and rotates are for word operands only and allow only single-bit shifts or rotates.

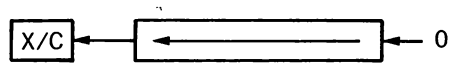
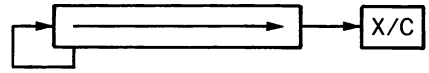
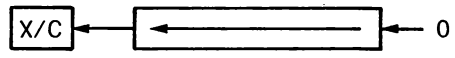
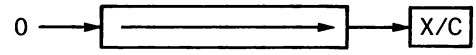
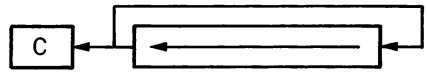
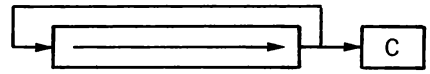
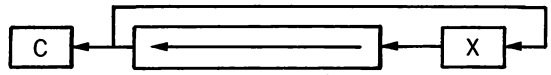
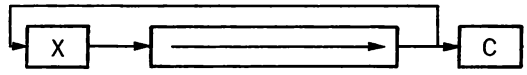
Table 3-4 is a summary of the shift and rotate operations.

Table 3-3. Logical Operations

Instruction	Operand Size	Operation
AND	8, 16, 32	Dn ∧ (EA) → Dn (EA) ∧ Dn → EA (EA) ∧ #xxx → EA
OR	8, 16, 32	Dn v (EA) → Dn (EA) v Dn → EA (EA) v #xxx → EA
EOR	8, 16, 32	(EA) ⊕ Dy → EA (EA) ⊕ #xxx → EA
NOT	8, 16, 32	~(EA) → EA

Note:
~ = invert

Table 3-4. Shift and Rotate Operations

Instruction	Operand Size	Operation
ASL	8, 16, 32	
ASR	8, 16, 32	
LSL	8, 16, 32	
LSR	8, 16, 32	
ROL	8, 16, 32	
ROR	8, 16, 32	
ROXL	8, 16, 32	
ROXR	8, 16, 32	

3.5 BIT MANIPULATION OPERATIONS

Bit manipulation operations are accomplished using the following instructions: bit test (BTST), bit test and set (BSET), bit test and clear (BCLR), and bit test and change (BCHG). Table 3-5 is a summary of the bit manipulation operations.

Table 3-5. Bit Manipulation Operations

Instruction	Operand Size	Operation
BTST	8, 32	~ bit of (EA) →Z
BSET	8, 32	~ bit of (EA) →Z 1 →bit of EA
BCLR	8, 32	~ bit of (EA) →Z 0 →bit of EA
BCHG	8, 32	~ bit of (EA) →Z ~ bit of (EA) →bit of EA

3.6 BINARY CODED DECIMAL OPERATIONS

Multiprecision arithmetic operations on binary coded decimal numbers are accomplished using the following instructions: add decimal with extend (ABCD), subtract decimal with extend (SBCD), and negate decimal with extend (NBCD). Table 3-6 is a summary of the binary coded decimal operations.

Table 3-6. Binary Coded Decimal Operations

Instruction	Operand Size	Operation
ABCD	8	$Dx_{10} + Dy_{10} + X \rightarrow Dx$ $Ax@_{-10} + Ay@_{-10} + X \rightarrow Ax@$
SBCD	8	$Dx_{10} - Dy_{10} - X \rightarrow Dx$ $Ax@_{-10} - Ay@_{-10} - X \rightarrow Ax@$
NBCD	8	$0 - (EA)_{10} - X \rightarrow EA$

3.7 PROGRAM CONTROL OPERATIONS

Program control operations are accomplished using a series of conditional and unconditional branch instructions and return instructions. These instructions are summarized in table 3-7.

The conditional instructions provide setting and branching for the following conditions:

CC — carry clear
CS — carry set
EQ — equal
F — never true
GE — greater or equal
GT — greater than
HI — high
LE — less or equal
LS — low or same
LT — less than
MI — minus
NE — not equal
PL — plus
T — always true
VC — no overflow
VS — overflow

Table 3-7. Program Control Operations

Instruction	Operation
Conditional	
Bcc	Branch conditionally (14 conditions) 8- and 16-bit displacement
DBcc	Test condition, decrement counter, and branch. 16-bit displacement
Scc	Set byte conditionally (16 conditions)
Unconditional	
BRA	Branch always 8- and 16-bit displacement
BSR	Branch to subroutine 8- and 16-bit displacement
JMP	Jump
JSR	Jump to subroutine
Returns	
RTR	Return and restore condition codes
RTS	Return from subroutine

3.8 SYSTEM CONTROL OPERATIONS

System control operations are accomplished by using privileged instructions, trap generating instructions, and instructions that use or modify the status register. These instructions are summarized in table 3-8.

Table 3-8. System Control Operations

Instruction	Operation
Privileged	
RESET	Reset external devices
RTE	Return from exception
STOP	Stop program execution
ORI to SR	Logical OR to status register
MOVE USP	Move user stack pointer
ANDI to SR	Logical AND to status register
EORI to SR	Logical EOR to status register
MOVE EA to SR	Load new status register
Trap Generating	
TRAP	Trap
TRAPV	Trap on overflow
CHK	Check register against bounds
Status Register	
ANDI to CCR	Logical AND to condition codes
EORI to CCR	Logical EOR to condition codes
MOVE EA to CCR	Load new condition codes
ORI to CCR	Logical OR to condition codes
MOVE SR to EA	Store status register

CHAPTER 4
SIGNAL AND BUS OPERATION DESCRIPTION

4.0 INTRODUCTION

This chapter contains a brief description of the input and output signals. A discussion of bus operation during the various machine cycles and operations is also given.

4.1 SIGNAL DESCRIPTION

The input and output signals can be functionally organized into the groups shown in figure 4-1. The following paragraphs provide a brief description of the signals and also a reference (if applicable) to other chapters that contain more detail about the function being performed.

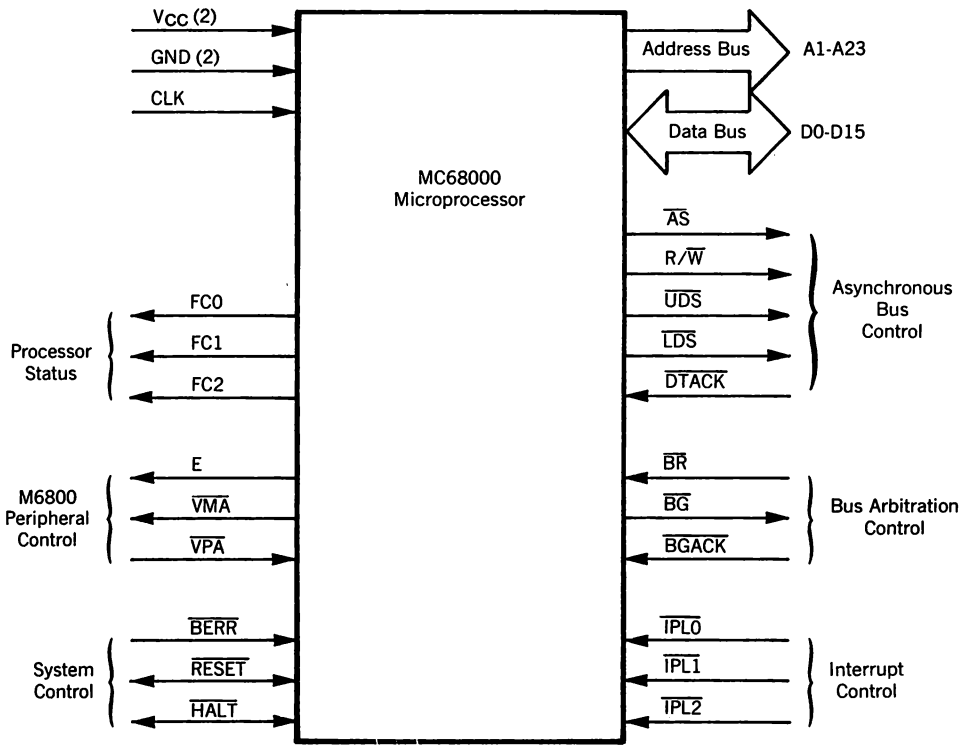


Figure 4-1. Input and Output Signals

4.1.1 ADDRESS BUS (A1 THROUGH A23). This 23-bit, unidirectional, three-state bus is capable of addressing 8 megawords of data. It provides the address for bus operation during all cycles except interrupt cycles. During interrupt cycles, address lines A1, A2, and A3 provide information about what level interrupt is being serviced while address lines A4 through A23 are all set to a logic high.

4.1.2 DATA BUS (D0 THROUGH D15). This 16-bit, bidirectional, three-state bus is the general purpose data path. It can transfer and accept data in either word or byte length. During an interrupt acknowledge cycle, data lines D0 through D7 contain the interrupt vector number.

4.1.3 ASYNCHRONOUS BUS CONTROL. Asynchronous data transfers are handled using the following control signals: address strobe, read/write, upper and lower data strobes, and data transfer acknowledge. These signals are explained in the following paragraphs.

4.1.3.1 Address Strobe ($\overline{AS}$). This signal indicates that there is a valid address on the address bus.

4.1.3.2 Read/Write (R/$\overline{W}$). This signal defines the data bus transfer as a read or write cycle. The R/$\overline{W}$ signal also works in conjunction with the upper and lower data strobes as explained in the following paragraph.

4.1.3.3 Upper And Lower Data Strobes ($\overline{UDS}$, $\overline{LDS}$). These signals control the data on the data bus as shown in table 4-1. When the R/$\overline{W}$ line is high, the processor will read from the data bus as indicated. When the R/$\overline{W}$ line is low, the processor will write to the data bus as shown.

Table 4-1. Data Strobe Control of Data Bus

$\overline{UDS}$	$\overline{LDS}$	R/$\overline{W}$	D8-D15	D0-D7
High	High	—	No valid data	No valid data
Low	Low	High	Valid data bits 8-15	Valid data bits 0-7
High	Low	High	No valid data	Valid data bits 0-7
Low	High	High	Valid data bits 8-15	No valid data
Low	Low	Low	Valid data bits 8-15	Valid data bits 0-7
High	Low	Low	Valid data bits 0-7*	Valid data bits 0-7
Low	High	Low	Valid data bits 8-15	Valid data bits 8-15*

*These conditions are a result of current implementation and may not appear on future devices.

4.1.3.4 Data Transfer Acknowledge ($\overline{\text{DTACK}}$). This input indicates that the data transfer is completed. When the processor recognizes $\overline{\text{DTACK}}$ during a read cycle, data is latched and the bus cycle terminated. When $\overline{\text{DTACK}}$ is recognized during a write cycle, the bus cycle is terminated.

4.1.4 BUS ARBITRATION CONTROL. These three signals form a bus arbitration circuit to determine what device will be the bus master device. Refer to paragraph 4.2.2 for a detailed description.

4.1.4.1 Bus Request ($\overline{\text{BR}}$). This input is wire ORed with all other devices that could be bus masters. This input indicates to the processor that some other device desires to become the bus master.

4.1.4.2 Bus Grant ($\overline{\text{BG}}$). This output indicates to all other potential bus master devices that the processor will release bus control at the end of the current bus cycle.

4.1.4.3 Bus Grant Acknowledge ($\overline{\text{BGACK}}$). This input indicates that some other device has become the bus master. This signal cannot be asserted until the following four conditions are met:

1. a bus grant has been received
2. address strobe is inactive which indicates that the microprocessor is not using the bus
3. data transfer acknowledge is inactive which indicates that either memory or the peripherals are not using the bus
4. bus grant acknowledge is inactive which indicates that no other device is still claiming bus mastership.

4.1.5 INTERRUPT CONTROL ($\overline{\text{IPL0}}$, $\overline{\text{IPL1}}$, $\overline{\text{IPL2}}$). These input pins indicate the encoded priority level of the device requesting an interrupt. Level seven is the highest priority while level zero indicates that no interrupts are requested. The least significant bit is given in $\overline{\text{IPL0}}$ and the most significant bit is contained in $\overline{\text{IPL2}}$. Refer to Chapter 5 for details on interrupt operation.

4.1.6 SYSTEM CONTROL. The system control inputs are used to either reset or halt the processor and to indicate to the processor that bus errors have occurred. The three system control inputs are explained in the following paragraphs.

4.1.6.1 Bus Error ($\overline{\text{BERR}}$). This input informs the processor that there is a problem with the cycle currently being executed. Problems are a result of:

1. nonresponding devices
2. interrupt vector number acquisition failure
3. illegal access request as determined by a memory management unit
4. other application dependent errors.

The bus error signal interacts with the halt signal to determine if exception processing should be performed or the current bus cycle should be retried.

Refer to paragraph 4.2.3 for additional information about the interaction of the bus error and halt signals.

4.1.6.2 Reset ($\overline{\text{RESET}}$). This bidirectional signal line acts to reset (initiate a system initialization sequence) the processor in response to an external reset signal. An internally generated reset (result of a RESET instruction) causes all external devices to be reset and the internal state of the processor is not affected. A total system reset (processor and external devices) is the result of external halt and reset signals applied at the same time. Refer to paragraph 4.2.4 for additional information about reset operation.

4.1.6.3 Halt ($\overline{\text{HALT}}$). When this bidirectional line is driven by an external device, it will cause the processor to stop at the completion of the current bus cycle. When the processor has been halted using this input, all control signals are inactive and all three-state lines are put in their high impedance state. Refer to paragraph 4.2.3 for additional information about the interaction between the halt and bus error signals.

When the processor has stopped executing instructions, such as in a double bus fault condition, the halt line is driven by the processor to indicate to external devices that the processor has stopped.

4.1.7 M6800 PERIPHERAL CONTROL. These control signals are used to allow the interfacing of synchronous M6800 peripheral devices with the asynchronous MC68000. These signals are explained in the following paragraphs.

4.1.7.1 Enable (E). This signal is the standard enable signal common to all M6800 type peripheral devices. The period for this output is ten MC68000 clock periods (six clocks low; four clocks high).

4.1.7.2 Valid Peripheral Address ($\overline{VPA}$). This input indicates that the device or region addressed is a M6800 family device and that data transfer should coincide with the enable (E) signal. This input also indicates that the processor should use automatic vectoring for an interrupt. Refer to Chapter 6.

4.1.7.3 Valid Memory Address ($\overline{VMA}$). This output is used to indicate to M6800 peripheral devices that there is a valid address on the address bus and the processor is synchronized to enable. This signal only responds to a valid peripheral address ($\overline{VPA}$) input which indicates that the peripheral is a M6800 family device.

4.1.8 PROCESSOR STATUS (FC0, FC1, FC2). These function code outputs indicate the mode (user or supervisor) and the cycle type currently being executed as shown in table 4-2. The information indicated by the function code outputs is valid whenever address strobe ($\overline{AS}$) is active.

Table 4-2. Function Code Outputs

FC0	FC1	FC2	Cycle Type
Low	Low	Low	(Undefined, Reserved)
Low	Low	High	User Data
Low	High	Low	User Program
Low	High	High	(Undefined, Reserved)
High	Low	Low	(Undefined, Reserved)
High	Low	High	Supervisor Data
High	High	Low	Supervisor Program
High	High	High	Interrupt Acknowledge

4.1.9 CLOCK (CLK). The clock input is a TTL compatible signal that is internally buffered for development of the internal clocks needed by the processor. The clock input shall be a constant frequency.

4.1.10 SIGNAL SUMMARY. Table 4-3 is a summary of all the signals discussed in the previous paragraphs.

Table 4-3. Signal Summary

Signal Name	Mnemonic	Input/Output	Active State	Three State
Address Bus	A1-A23	output	high	yes
Data Bus	D0-D15	input/output	high	yes
Address Strobe	$\overline{AS}$	output	low	yes
Read/Write	R/$\overline{W}$	output	read-high write-low	yes
Upper and Lower Data Strobes	$\overline{UDS}$, $\overline{LDS}$	output	low	yes
Data Transfer Acknowledge	$\overline{DTACK}$	input	low	no
Bus Request	$\overline{BR}$	input	low	no
Bus Grant	$\overline{BG}$	output	low	no
Bus Grant Acknowledge	$\overline{BGACK}$	input	low	no
Interrupt Priority Level	$\overline{IPL0}$, $\overline{IPL1}$, $\overline{IPL2}$	input	low	no
Bus Error	$\overline{BERR}$	input	low	no
Reset	$\overline{RESET}$	input/output	low	no*
Halt	$\overline{HALT}$	input/output	low	no*
Enable	E	output	—	no
Valid Memory Address	$\overline{VMA}$	output	low	yes
Valid Peripheral Address	$\overline{VPA}$	input	low	no
Function Code Output	FC0, FC1, FC2	output	high	yes
Clock	CLK	input	high	no
Power Input	V_{CC}	Input	—	—
Ground	GND	Input	—	—

*open drain

4.2 BUS OPERATION

The following paragraphs explain control signal and bus operation during data transfer operations, bus arbitration, bus error and halt conditions, and reset operation.

4.2.1 DATA TRANSFER OPERATIONS. Transfer of data between devices involves the following leads:

- Address Bus A1 through A23
- Data Bus D0 through D15
- Control Signals

The address and data buses are separate parallel buses used to transfer data using an asynchronous bus structure. In all cycles, the bus master assumes responsibility for deskewing all signals it issues at both the start and end of a cycle. In addition, the bus

master is responsible for deskewing the acknowledge and data signals from the slave device.

The following paragraphs explain the read, write, and read-modify-write cycles. The indivisible read-modify-write cycle is the method used by the MC68000 for interlocked multiprocessor communications.

NOTE

The terms **assertion** and **negation** will be used extensively. This is done to avoid confusion when dealing with a mixture of "active-low" and "active-high" signals. The term assert or assertion is used to indicate that a signal is active or true independent of whether that voltage is low or high. The term negate or negation is used to indicate that a signal is inactive or false.

Each of the bus cycle operations is defined in terms of a succession of states. These states are described for reference purposes only, and do not necessarily correspond to any implemented machine states.

4.2.1.1 Read Cycle. During a read cycle, the processor receives data from memory or a peripheral device. The processor reads bytes of data in all cases. If the instruction specifies a word (or double word) operation, the processor reads both bytes. When the instruction specifies byte operation, the processor uses the internal A0 bit to determine which byte to read and then issues the data strobe required for that byte. For byte operations, when the A0 bit equals zero, the upper data strobe is issued. When the A0 bit equals one, the lower data strobe is issued. When the data is received, the processor correctly positions it internally.

A word read cycle flow chart is given in figure 4-2. A byte read cycle flow chart is given in figure 4-3. Read cycle timing is given in figure 4-4 and figure 4-5 details word and byte read cycle operation. Refer to these illustrations during the following detailed discussion.

At state zero (S0) in the read cycle, the address bus (A1 through A23) and the function code outputs (FC0 through FC2) are in the high impedance state. The read/write (R/$\overline{W}$) signal is switched high to indicate a read cycle. One half clock cycle later, at state 1, the address bus and function code outputs are released from the high impedance state. The function code outputs indicate which address space that this cycle will operate on.

In state 2, the address strobe ($\overline{AS}$) is asserted to indicate that there is a valid address on the address bus and the upper and lower data strobe ($\overline{UDS}$, $\overline{LDS}$) is asserted as required. The memory or peripheral device uses the address bus and the address strobe to determine if it has been selected. The selected device uses the read/write signal and the data strobe to place its information on the data bus. Concurrent with placing data on the data bus, the selected device asserts data transfer acknowledge ($\overline{DTACK}$). No new control signals are issued during states 3 and 4.

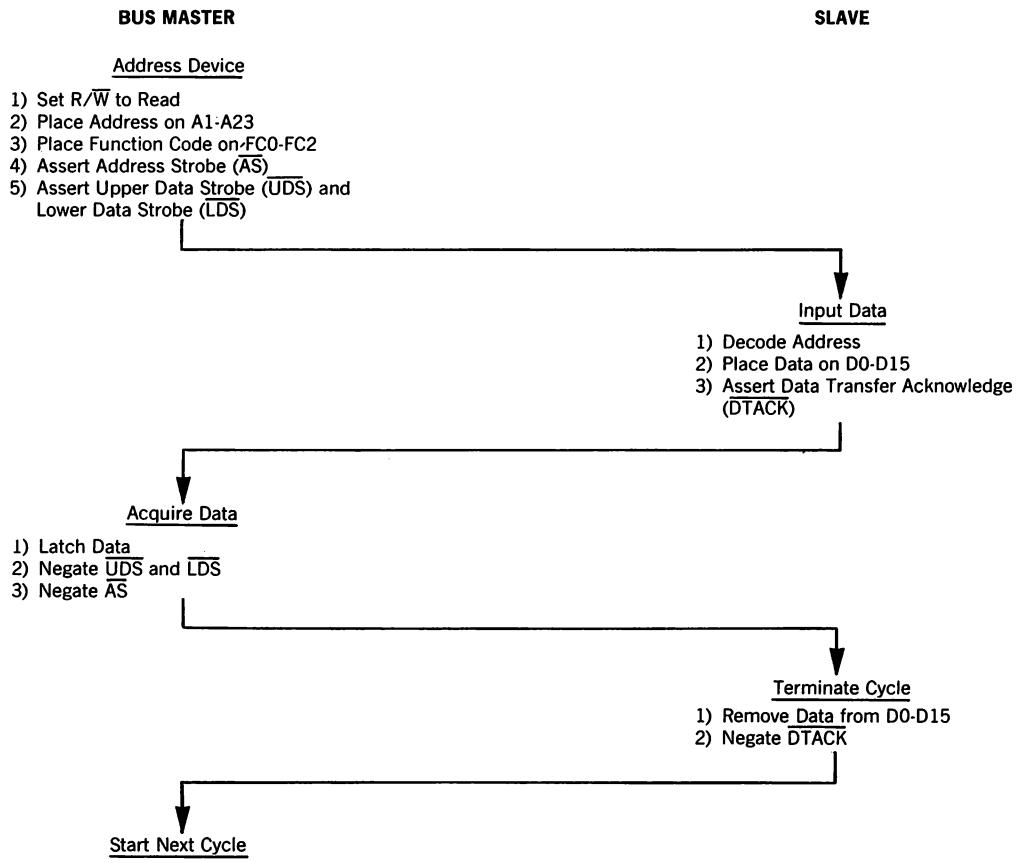

BUS MASTER
SLAVE
Address Device
1) Set R/W to Read
2) Place Address on A1-A23
3) Place Function Code on FC0-FC2
4) Assert Address Strobe (AS)
5) Assert Upper Data Strobe (UDS) and Lower Data Strobe (LDS)
Input Data
1) Decode Address
2) Place Data on D0-D15
3) Assert Data Transfer Acknowledge (DTACK)
Acquire Data
1) Latch Data
2) Negate UDS and LDS
3) Negate AS
Terminate Cycle
1) Remove Data from D0-D15
2) Negate DTACK
Start Next Cycle

Figure 4-2. Word Read Cycle Flow Chart

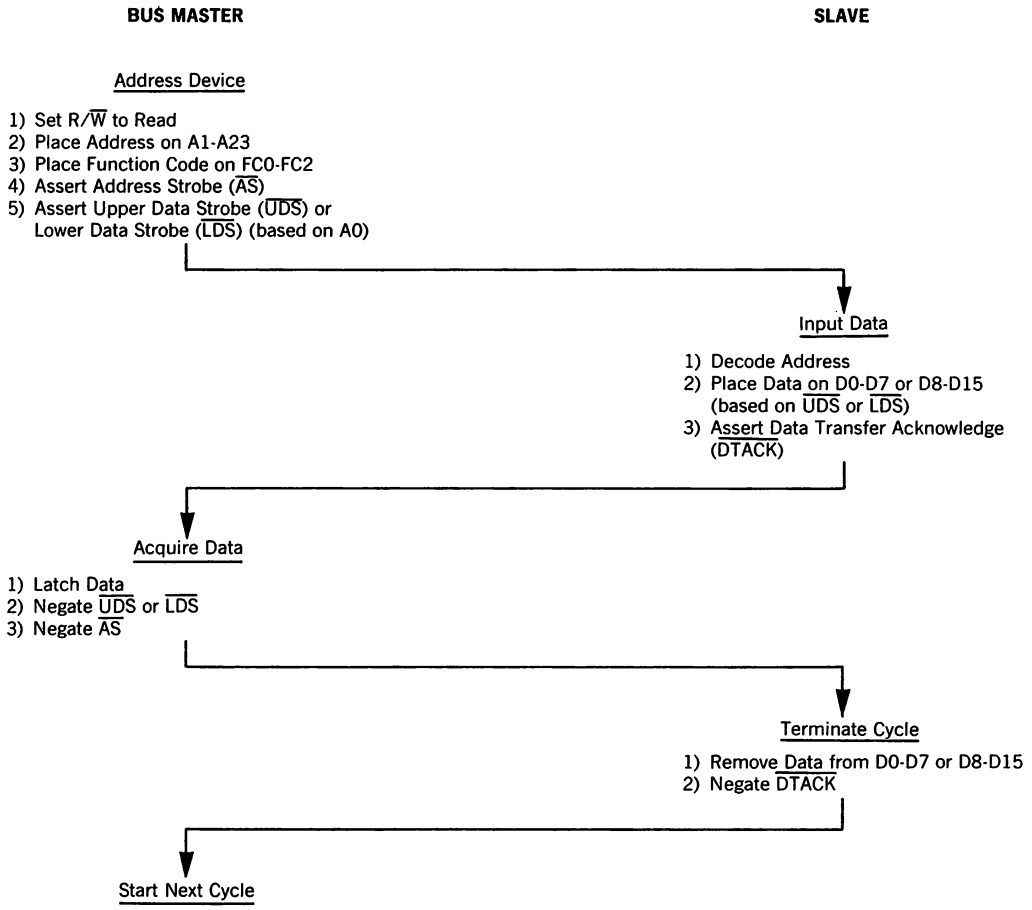

BUS MASTER
SLAVE
Address Device
1) Set R/W to Read
2) Place Address on A1-A23
3) Place Function Code on FC0-FC2
4) Assert Address Strobe (AS)
5) Assert Upper Data Strobe (UDS) or
Lower Data Strobe (LDS) (based on A0)
Input Data
1) Decode Address
2) Place Data on D0-D7 or D8-D15
(based on UDS or LDS)
3) Assert Data Transfer Acknowledge
(DTACK)
Acquire Data
1) Latch Data
2) Negate UDS or LDS
3) Negate AS
Terminate Cycle
1) Remove Data from D0-D7 or D8-D15
2) Negate DTACK
Start Next Cycle

Figure 4-3. Byte Read Cycle Flow Chart

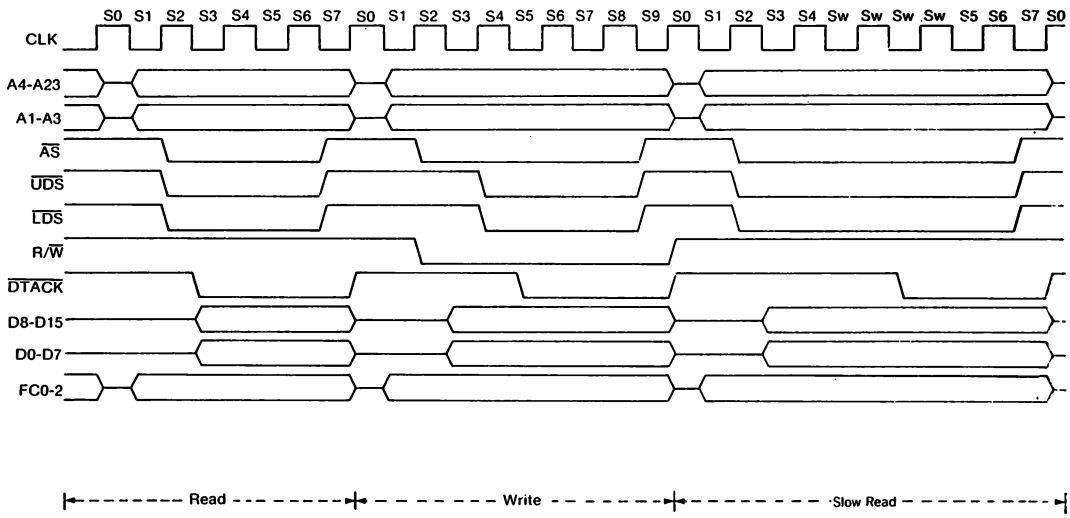


Figure 4-4. Read and Write Cycle Timing Diagram

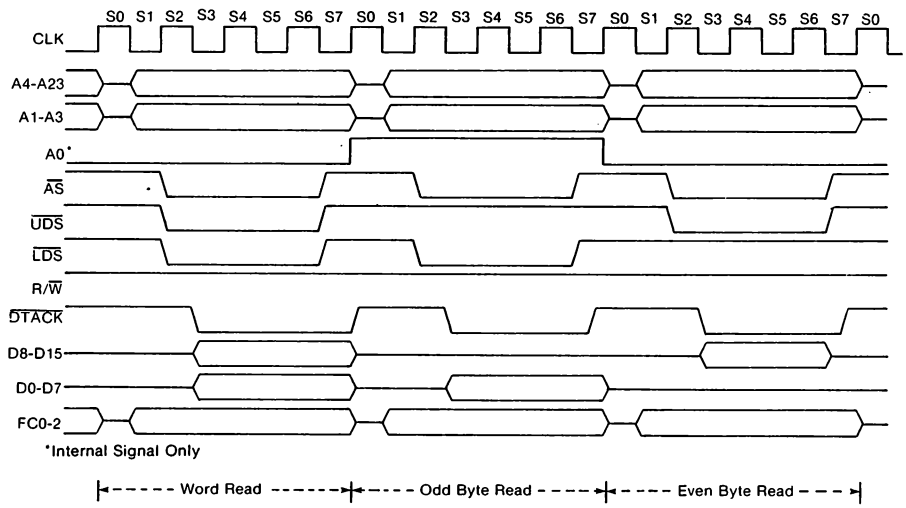


Figure 4-5. Word and Byte Read Cycle Timing Diagram

Data transfer acknowledge must be present at the processor at the start of state 5 or the processor will substitute wait states for states 5 and 6. State 5 starts the synchronization of the returning data transfer acknowledge. The bus interface circuitry issues request for subsequent internal cycles during state 6. At the end of state 6 (beginning of state 7) incoming data is latched into an internal data bus holding register.

During state 7, address strobe and the upper and/or lower data strobes are negated. The address bus is held valid through state 7 to allow for static memory operation and signal skew. The read/write signal and the function code outputs also remain valid through state 7 to ensure a correct transfer operation. The slave device keeps its data asserted until it detects the negation of either the address strobe or the upper and/or lower data strobe. The slave device must remove its data and data transfer acknowledge within one clock period of recognizing the negation of the address or data strobes. Note that the data bus may not become free and data transfer acknowledge may not be removed until state 0 or 1.

When address strobe is negated, the slave device is released. Note that a slave device must remain selected as long as address strobe is asserted to ensure the correct functioning of the read-modify-write cycle as explained in paragraph 4.2.1.3.

4.2.1.2 Write Cycle. During a write cycle, the processor sends data to memory or a peripheral device. The processor writes bytes of data in all cases. If the instruction specifies a word operation, the processor writes both bytes. When the instruction specifies a byte operation, the processor uses the internal A0 bit to determine which byte to write and then issues the data strobe required for that byte. For byte operations, when the A0 bit equals zero, the upper data strobe is issued. When the A0 bit equals one, the lower data strobe is issued. A word write cycle flow chart is given in figure 4-6. A byte write cycle flow chart is given in figure 4-7. Write cycle timing is given in figure 4-4 and figure 4-8 details word and byte write cycle operation. Refer to these illustrations during the following detailed discussion.

At state zero (S0) in the write cycle, the address bus (A1 through A23) and the function code outputs (FC0 through FC2) are in the high impedance state.

NOTE

The read/write ($R/\overline{W}$) signal remains high until state 2 to prevent bus conflicts with preceding read cycles. The data bus is not driven until state 3.

One half clock later, at state 1, the address bus and the function code outputs are released from the high impedance state. The function code outputs indicate which address space that this cycle will operate on.

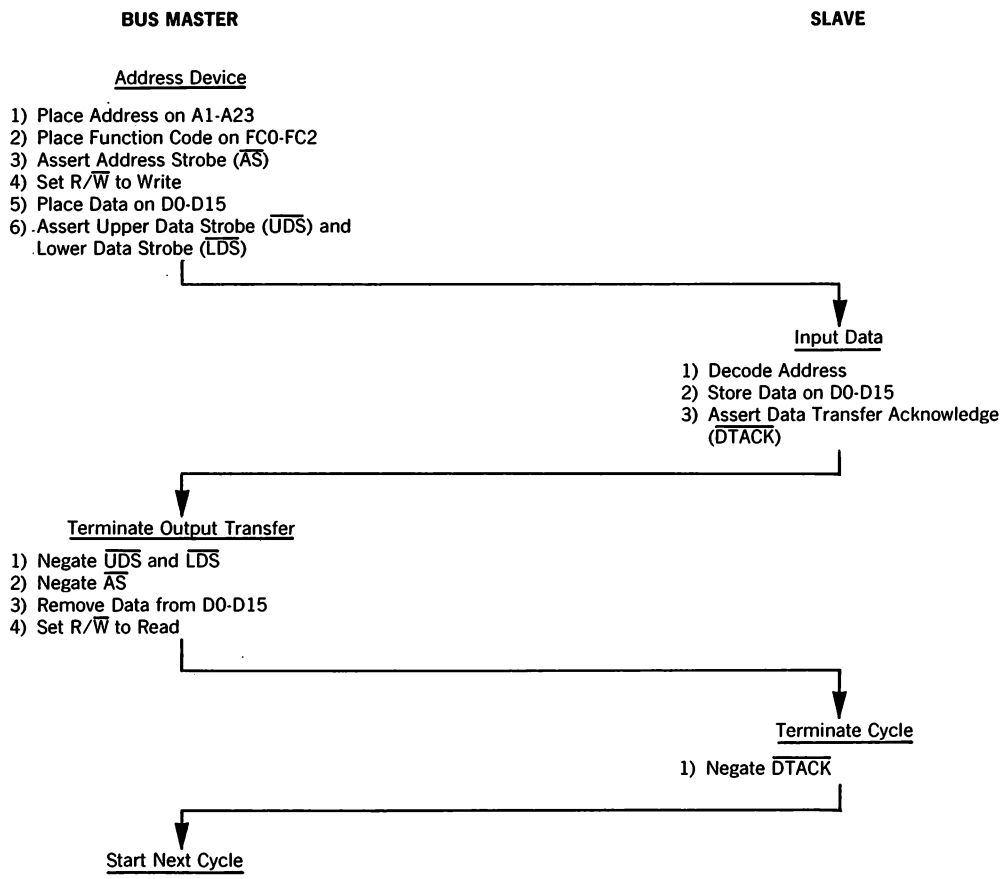

BUS MASTER
SLAVE
Address Device
1) Place Address on A1-A23
2) Place Function Code on FC0-FC2
3) Assert Address Strobe (AS)
4) Set R/W to Write
5) Place Data on D0-D15
6) Assert Upper Data Strobe (UDS) and Lower Data Strobe (LDS)
Input Data
1) Decode Address
2) Store Data on D0-D15
3) Assert Data Transfer Acknowledge (DTACK)
Terminate Output Transfer
1) Negate UDS and LDS
2) Negate AS
3) Remove Data from D0-D15
4) Set R/W to Read
Terminate Cycle
1) Negate DTACK
Start Next Cycle

Figure 4-6. Word Write Cycle Flow Chart

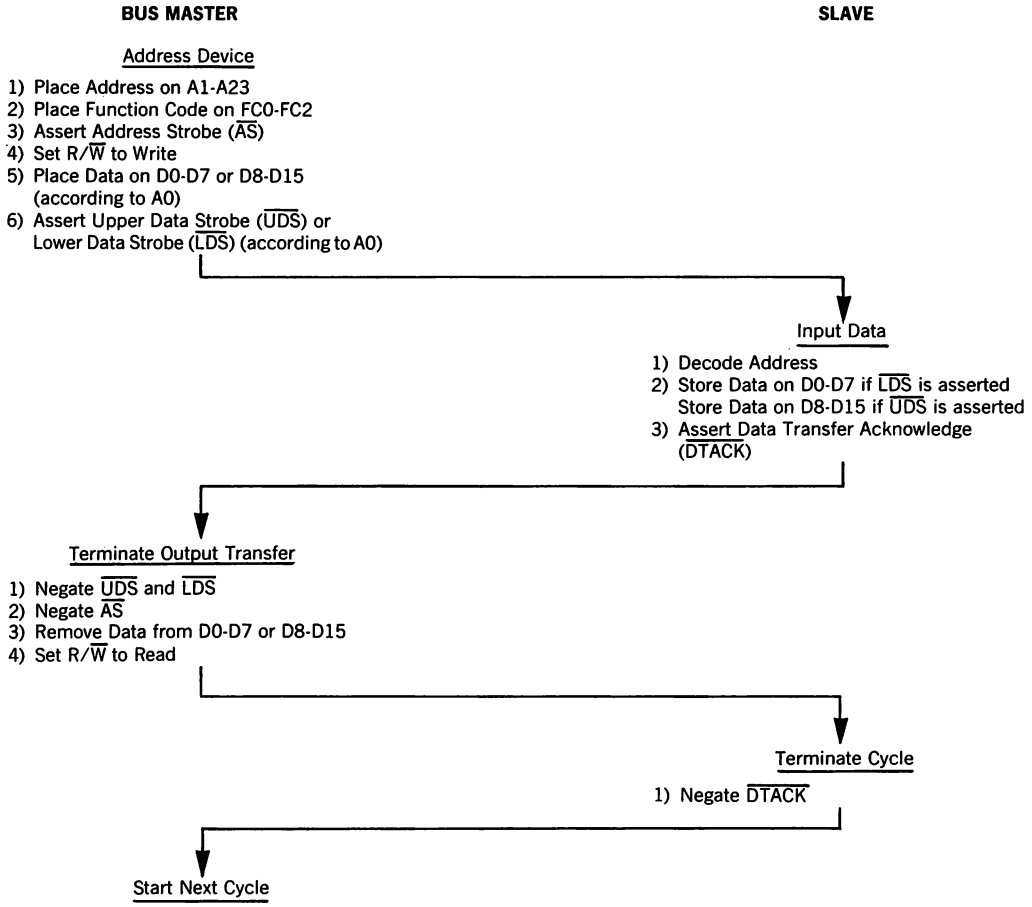

BUS MASTER
SLAVE
Address Device
1) Place Address on A1-A23
2) Place Function Code on FC0-FC2
3) Assert Address Strobe (AS)
4) Set R/W to Write
5) Place Data on D0-D7 or D8-D15 (according to A0)
6) Assert Upper Data Strobe (UDS) or Lower Data Strobe (LDS) (according to A0)
Input Data
1) Decode Address
2) Store Data on D0-D7 if LDS is asserted
Store Data on D8-D15 if UDS is asserted
3) Assert Data Transfer Acknowledge (DTACK)
Terminate Output Transfer
1) Negate UDS and LDS
2) Negate AS
3) Remove Data from D0-D7 or D8-D15
4) Set R/W to Read
Terminate Cycle
1) Negate DTACK
Start Next Cycle

Figure 4-7. Byte Write Cycle Flow Chart

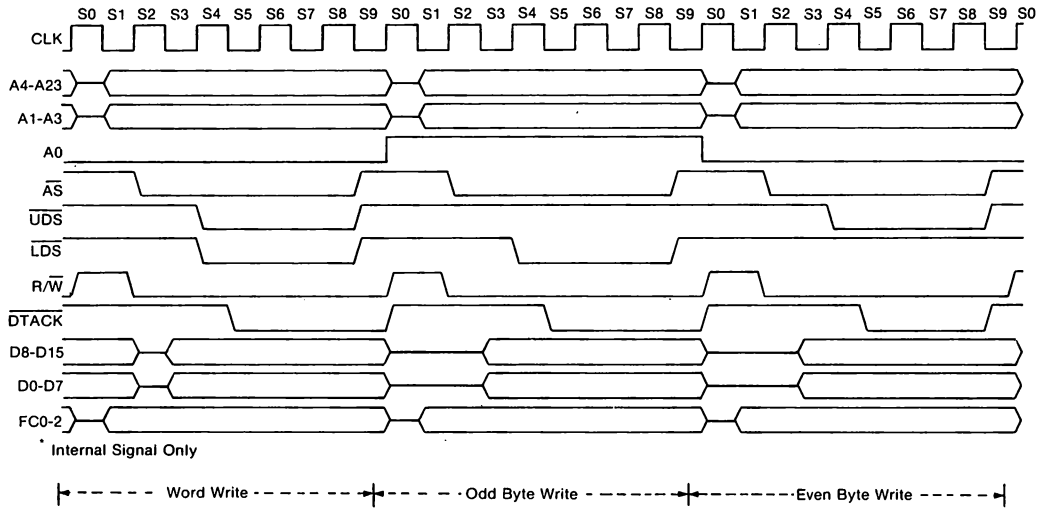


Figure 4-8. Word and Byte Write Cycle Timing Diagram

In state 2, the address strobe ($\overline{AS}$) is asserted to indicate that there is a valid address on the address bus. The memory or peripheral device uses the address bus and the address strobe to determine if it has been selected. During state 2, the read/write signal is switched low to indicate a write cycle. When external processor data bus buffers are required, the read/write line provides sufficient directional control. Data is not asserted during this state to allow sufficient turn around time for external data buffers (if used). Data is asserted onto the data bus during state 3.

In state 4, the data strobes are asserted as required to indicate that the data bus is stable. The selected device uses the read/write signal and the data strobes to take its information from the data bus. The selected device asserts data transfer acknowledge ($\overline{DTACK}$) when it has successfully stored the data. No new control signals are issued during states 5 and 6.

Data transfer acknowledge must be present at the processor at the start of state 7 or the processor will substitute wait states for states 7 and 8. State 7 starts the synchonization of the returning data transfer acknowledge. The bus interface circuitry issues requests for subsequent internal cycles during state 8.

During state 9, address strobe and the upper and/or lower data strobes are negated. The address and data buses are held valid through state 9 to allow for static memory operation and signal skew. The read/write signal and the function code outputs also remain valid through state 9 to ensure a correct transfer operation. The slave device keeps its data transfer acknowledge asserted until it detects the negation of either the address strobe or the upper and/or lower data strobe. The slave device must remove its data transfer acknowledge within one clock period after recognizing the negation of the address or data strobes. Note that the processor releases the data bus at the end of

state 9 but that data transfer acknowledge may not be removed until state 0 or 1.

When address strobe is negated, the slave device is released.

4.2.1.3 Read-Modify-Write Cycle. The read-modify-write cycle performs a read, modifies the data in the arithmetic-logic unit, and writes the data back to the same address. In the MC68000 this cycle is indivisable in that the address strobe is asserted throughout the entire cycle. The test and set (TAS) instruction uses this cycle to provide meaningful communication between processors in a multiple processor environment. This instruction is the only instruction that uses the read-modify-write cycle and since the test and set instruction only operates on bytes, all read-modify-write cycles are byte operations. A read-modify-write cycle flow chart is given in figure 4-9 and a timing diagram is given in figure 4-10. Refer to these illustrations during the following detailed discussions.

At state zero (S0) in the read-modify-write cycle, the address bus (A1 through A23) and function code outputs (FC0 through FC2) are in the high impedance state. The read/write (R/$\overline{W}$) signal is switched high to indicate a read cycle. One half clock cycle later, at state 1, the address bus and the function code outputs are released from the high impedance state. The function code outputs indicate which address space that this cycle will operate on.

In state 2, the address strobe ($\overline{AS}$) is asserted to indicate that there is a valid address on the address bus and the upper or lower data strobe ($\overline{UDS}$, $\overline{LDS}$) is asserted as required. The memory or peripheral device uses the address bus and the address strobe to determine if it has been selected. The selected device uses the read/write signal and the data strobe to place its information on the data bus. Concurrent with placing data on the data bus, the selected device asserts data transfer acknowledge ($\overline{DTACK}$). No new control signals are issued during states 3 and 4.

Data transfer acknowledge must be present at the processor at the start of state 5 or the processor will substitute wait states for states 5 and 6. State 5 starts the synchronization of the returning data transfer acknowledge. The bus interface circuitry issues requests for subsequent internal cycles during state 6. At the end of state 6 (beginning of state 7) incoming data is latched into an internal data bus holding register.

During state 7, address strobe and the upper or lower data strobe is negated. The address bus, address strobe, read/write signal, and function code outputs remain as they were in preparation for the write portion of the cycle. The slave device keeps its data asserted until it detects the negation of the upper or lower data strobe. The slave device must remove its data and data transfer acknowledge within one clock period of recognizing the negation of the data strobes. No new control signals are issued during state 8. Internal modification of data may occur at this time.

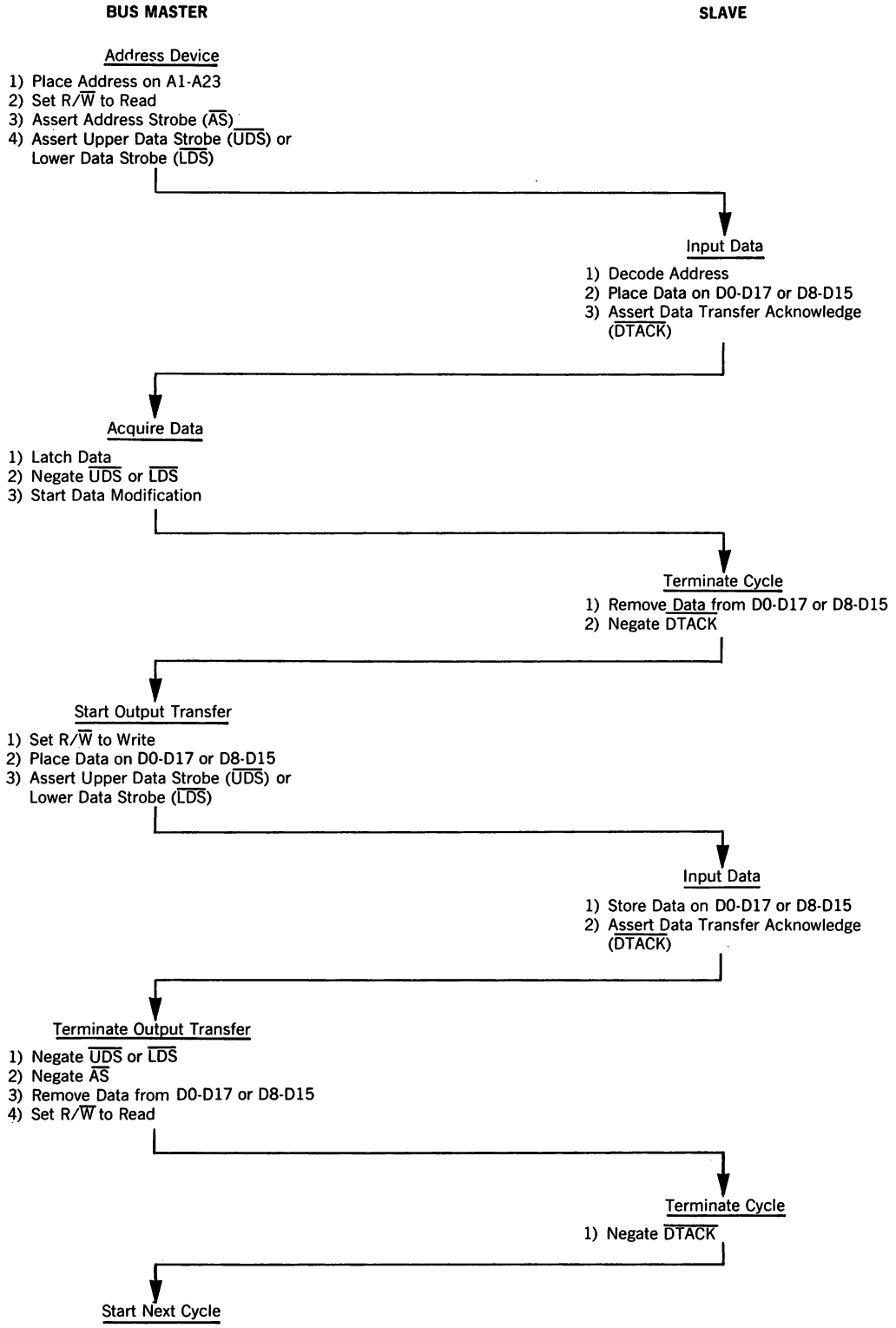

BUS MASTER
SLAVE
Address Device
1) Place Address on A1-A23
2) Set R/W to Read
3) Assert Address Strobe (AS)
4) Assert Upper Data Strobe (UDS) or Lower Data Strobe (LDS)
Input Data
1) Decode Address
2) Place Data on D0-D17 or D8-D15
3) Assert Data Transfer Acknowledge (DTACK)
Acquire Data
1) Latch Data
2) Negate UDS or LDS
3) Start Data Modification
Terminate Cycle
1) Remove Data from D0-D17 or D8-D15
2) Negate DTACK
Start Output Transfer
1) Set R/W to Write
2) Place Data on D0-D17 or D8-D15
3) Assert Upper Data Strobe (UDS) or Lower Data Strobe (LDS)
Input Data
1) Store Data on D0-D17 or D8-D15
2) Assert Data Transfer Acknowledge (DTACK)
Terminate Output Transfer
1) Negate UDS or LDS
2) Negate AS
3) Remove Data from D0-D17 or D8-D15
4) Set R/W to Read
Terminate Cycle
1) Negate DTACK
Start Next Cycle

Figure 4-9. Read-Modify-Write Cycle Flow Chart

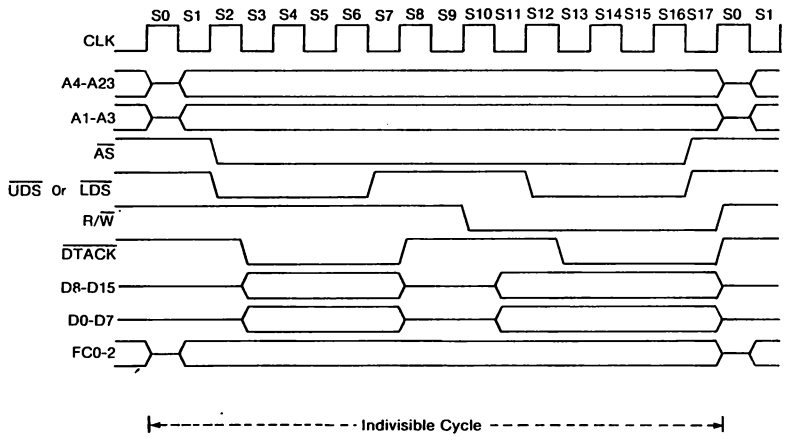


Figure 4-10. Read-Modify-Write Cycle Timing Diagram

NOTE

The read/write signal remains high until state 10 to prevent bus conflicts with the preceding read portion of the cycle and the data bus is not asserted by the processor until state 11.

No new control signals are issued in state 9.

In state 10, the read/write signal is switched low to indicate a write cycle. When external processor data bus buffers are required, the read/write line provides sufficient directional control. Data is not asserted during this state to allow sufficient turn around time for external data buffers (if used). Data is asserted onto the data bus during state 11.

In state 12, the data strobe is asserted as required to indicate that the data bus is stable. The selected device uses the read/write signal and the data strobe to take its information from the data bus. The selected device asserts data transfer acknowledge ($\overline{DTACK}$) when it has successfully stored its data. No new control signals are issued during states 13 and 14.

Data transfer acknowledge must be present at the processor at the start of state 15 or the processor will substitute wait states for states 15 and 16. State 15 starts the synchronization of the returning data transfer acknowledge for the write portion of the cycle. The bus interface circuitry issues requests for subsequent internal cycles during state 16.

During state 17, address strobe and the upper or lower data strobe is negated. The address and data buses are held valid through state 17 to allow for static memory operation and signal skew. The read/write signal and the function code outputs also remain valid through state 17 to ensure a correct transfer operation. The slave device keeps its data transfer acknowledge asserted until it detects the negation of either the address strobe or the upper or lower data strobe. The slave device must remove its data transfer acknowledge within once clock period after recognizing the negation of the address or data strobes. Note that the processor releases the data bus at the end of state 17 but that data transfer acknowledge may not be removed until state 0 or 1. When address strobe is negated the slave device is released.

4.2.2 BUS ARBITRATION. Bus arbitration is a technique used by master type devices to request, be granted, and acknowledge bus mastership. In its simplest form, it consists of:

1. Asserting a bus mastership request.
2. Receiving a grant that the bus is available at the end of the current cycle.
3. Acknowledging that mastership has been assumed.

Figure 4-11 is a flow chart showing the detail involved in a request from a single device. Figure 4-12 is a timing diagram for the same operations. This technique allows processing of bus requests during data transfer cycles.

The timing diagram shows that the bus request is negated at the time that an acknowledge is asserted. This type of operation would be true for a system consisting of the processor and one device capable of bus mastership. In systems having a number of devices capable of bus mastership, the bus request line from each device is wire ORed to the processor. In this system, it is easy to see that there could be more than one bus request being made. The timing diagram shows that the bus grant signal is negated a few clock cycles after the transition of the acknowledge ($\overline{\text{BGACK}}$) signal.

However, if the bus requests are still pending, the processor will assert another bus grant within a few clock cycles after it was negated. This additional assertion of bus grant allows external arbitration circuitry to select the next bus master before the current bus master has completed its requirements. The following paragraphs provide additional information about the three steps in the arbitration process.

4.2.2.1 Requesting the Bus. External devices capable of becoming bus masters request the bus by asserting the bus request ($\overline{\text{BR}}$) signal. This is a wire ORed signal (although it need not be constructed from open collector devices) that indicates to the processor that some external device requires control of the external bus. The processor is effectively at a lower bus priority level than the external device and will relinquish the bus after it has completed the last bus cycle it has started.

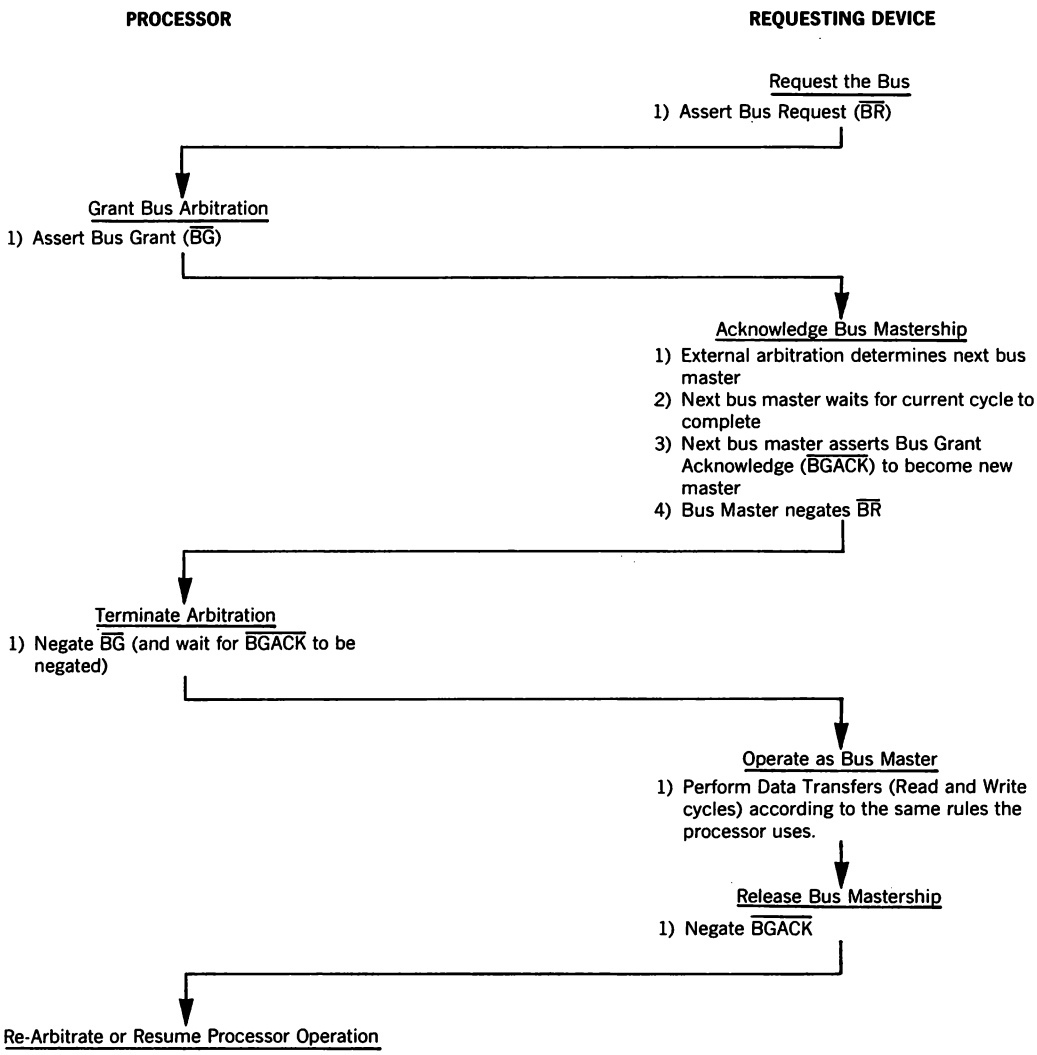

PROCESSOR
REQUESTING DEVICE
Request the Bus
1) Assert Bus Request (BR)
Grant Bus Arbitration
1) Assert Bus Grant (BG)
Acknowledge Bus Mastership
1) External arbitration determines next bus master
2) Next bus master waits for current cycle to complete
3) Next bus master asserts Bus Grant Acknowledge (BGACK) to become new master
4) Bus Master negates BR
Terminate Arbitration
1) Negate BG (and wait for BGACK to be negated)
Operate as Bus Master
1) Perform Data Transfers (Read and Write cycles) according to the same rules the processor uses.
Release Bus Mastership
1) Negate BGACK
Re-Arbitrate or Resume Processor Operation

Figure 4-11. Bus Arbitration Cycle Flow-Chart

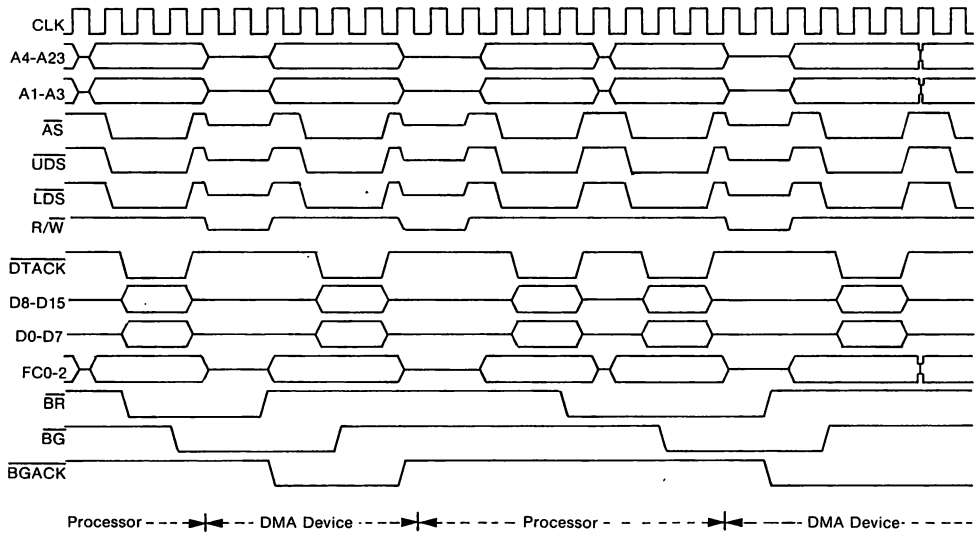


Figure 4-12. Bus Arbitration Cycle Timing Diagram

When no acknowledge is received before the bus request signal goes inactive, the processor will continue processing when it detects that bus request is inactive. This allows ordinary processing to continue if the arbitration circuitry responded to noise inadvertently.

4.2.2.2 Receiving the Bus Grant. The processor issues bus grant ($\overline{BG}$) as soon as possible. Normally this is immediately after internal synchronization. The only exception to this occurs when the processor has made an internal decision to execute the next bus cycle but has not progressed far enough into the cycle to have asserted the address strobe ($\overline{AS}$) signal. In this case, bus grant will not be asserted until one clock after address strobe is asserted to indicate to external devices that a bus cycle is being executed.

The bus grant signal may be routed through a daisy-chained network or through a specific priority-encoded network. The processor is not affected by the external method of arbitration as long as the protocol is obeyed.

4.2.2.3 Acknowledgement of Mastership. Upon receiving a bus grant, the requesting device waits until address strobe, data transfer acknowledge, and bus grant acknowledge are negated before issuing its own $\overline{BGACK}$. The negation of the address strobe indicates that the previous master has completed its cycle, the negation of bus grant acknowledge indicates that the previous master has released the bus. (While address strobe is asserted no device is allowed to "break into" a cycle). The negation of data transfer acknowledge indicates the previous slave has

terminated its connection to the previous master. Note that in some applications data transfer acknowledge might not enter into this function. General purpose devices would then be connected such that they were only dependent on address strobe. When bus grant acknowledge is issued the device is bus master until it negates bus grant acknowledge. Bus grant acknowledge should not be negated until after the bus cycle(s) is (are) completed. Bus mastership is terminated at the negation of bus grant acknowledge.

The bus request from the granted device should be dropped when bus grant acknowledge is asserted. If bus request is still asserted after bus grant acknowledge is negated the processor performs another arbitration sequence and issues another bus grant. Note that the processor does not perform any external bus cycles before it re-asserts bus grant.

4.2.3 BUS ERROR AND HALT OPERATION. In a bus architecture that requires a handshake from an external device, the possibility exists that the handshake might not occur. Since different systems will require different maximum response time, a bus error input is provided. External circuitry must be used to determine the duration between address strobe and data transfer acknowledge before issuing a bus error signal. When a bus error signal is received, the processor has two options: initiate a bus error exception sequence or try running the bus cycle again.

4.2.3.1 Exception Sequence. The bus error exception sequence is entered when the processor receives a bus error signal and the halt pin is inactive. Figure 4-13 is a timing diagram for the exception sequence. The sequence is composed of the following elements:

1. Stacking the program counter and status register.
2. Stacking the error information
3. Reading the bus error vector table entry
4. Executing the bus error handler routine

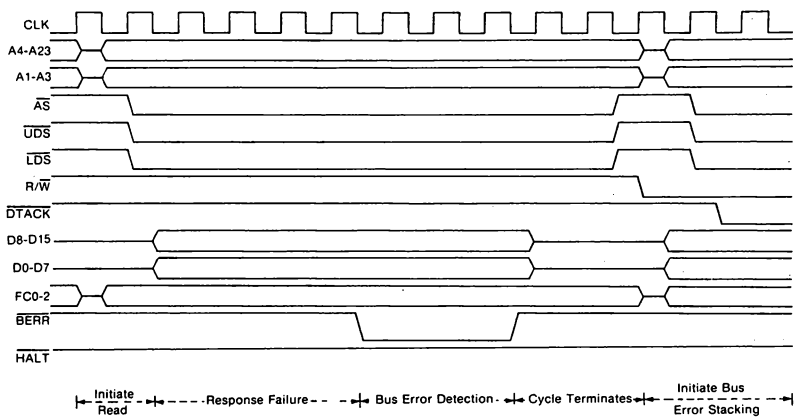


Figure 4-13. Bus Error Timing Diagram

The stacking of the program counter and the status register is the same as if an interrupt had occurred. Several additional items are stacked when a bus error occurs. These items are used to determine the nature of the error and to correct it, if possible. The bus error vector is vector number two located at address $000008. The processor loads the new program counter from this location. A software bus error handler routine is then executed by the processor. Refer to Chapter 5 for additional information.

4.2.3.2 Re-Running the Bus Cycle. When the processor receives a bus error signal and the halt pin is being driven by an external device, the processor enters the re-run sequence. Figure 4-14 is a timing diagram for re-running of the bus cycle.

The processor completes the bus cycle, then puts the address, data, function code, and control leads in the high impedance state. The processor remains "halted," and will not run another bus cycle until the halt signal is removed by external logic. Then the processor will re-run the previous bus cycle, using the same address, the same function codes, the same data (for a write operation), and the same controls. The bus error signal should be removed before the halt signal is removed.

NOTE:

The processor will not re-run a read-modify-write cycle. This restriction is made to guarantee that the entire cycle runs correctly and that the write operation of a Test-and-Set operation is performed without ever releasing $\overline{AS}$.

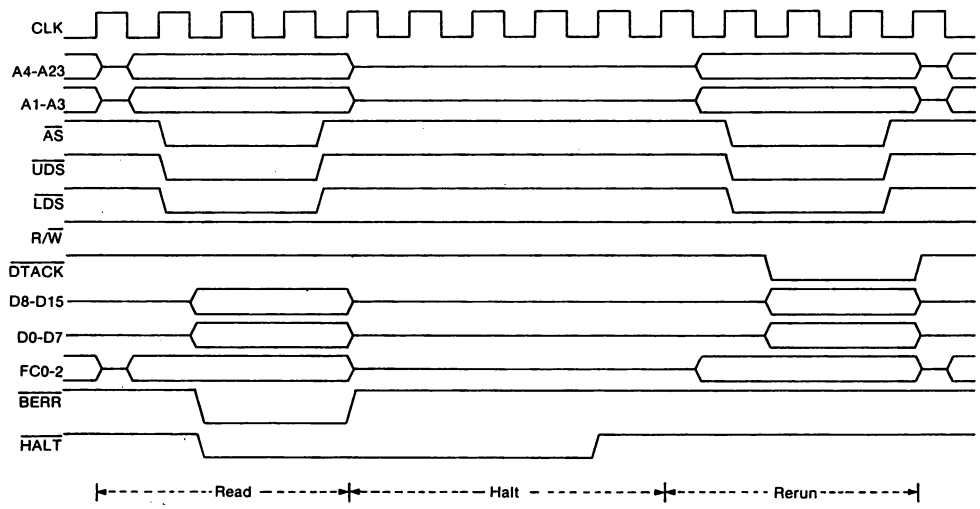


Figure 4-14. Re-Run Bus Cycle Timing Information

4.2.3.3 Halt Operation with No Bus Error. The halt input signal to the MC68000 performs a Halt/Run/Single-Step function in a similar fashion to the M6800 halt function. The halt and run modes are somewhat self explanatory in that when the halt signal is constantly active the processor "halts" (does nothing) and when the halt signal is constantly inactive the processor "runs" (does something).

The single-step mode is derived from correctly timed transitions on the halt signal input. It forces the processor to execute a single bus cycle by entering the "run" mode until the processor starts a bus cycle then changing to the "halt" mode. Thus, the single step mode allows the user to proceed through (and therefore debug) processor operations one bus cycle at a time.

Figure 4-15 details the timing required for correct single-step operation and figure 4-16 shows a simple circuit for providing the single-step function. Some care must be exercised to avoid harmful interactions between the bus error signal and the halt pin when using the single cycle mode as a debugging tool. This is also true of interactions between the halt and reset lines since these can reset the machine.

When the processor completes a bus cycle after recognizing that the halt signal is active, most three-state signals are put in the high impedance state. These include:

1. address lines
2. data lines
3. function code lines

This is required for correct performance of the re-run bus cycle operation described in paragraph 4.2.3.2.

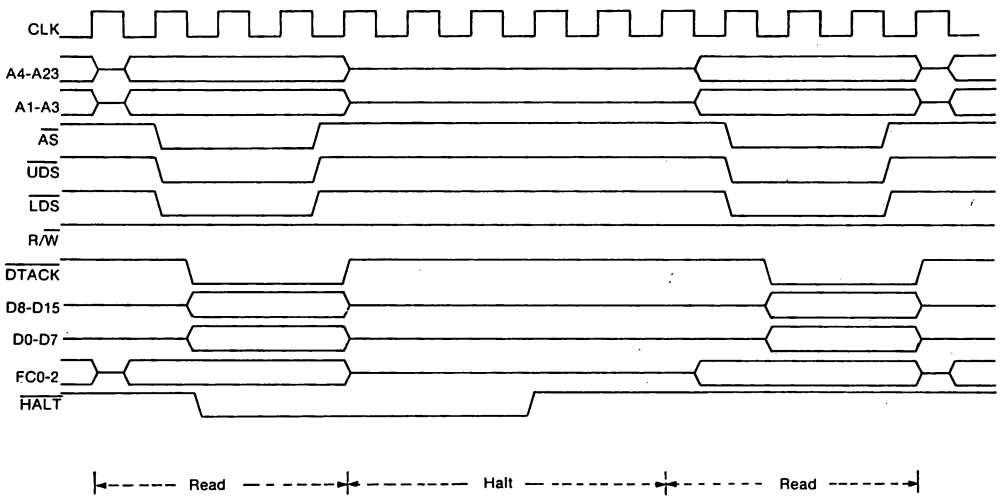


Figure 4-15. Halt Signal Timing Characteristics

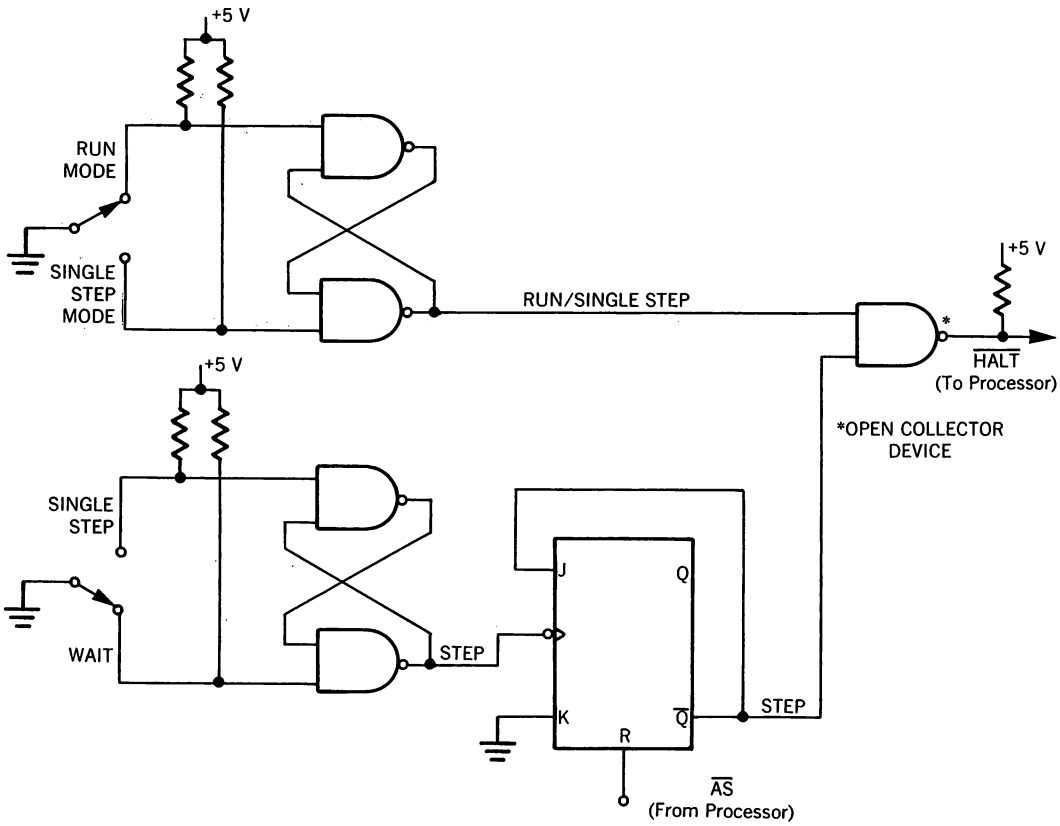


Figure 4-16. Simplified Single-Step Circuit

Note that when the processor honors a request to halt, the function codes are put in the high impedance state (their buffer characteristics are the same as the address buffers). While the processor is honoring the halt request, bus arbitration performs as usual. That is, halting has no effect on bus arbitration. It is the bus arbitration function that removes the control signals from the bus.

The halt function and the hardware trace capability allow the hardware debugger to trace single bus cycles or single instructions at a time. These processor capabilities, along with a software debugging package, give total debugging flexibility.

4.2.3.4 Double Bus Faults. When a bus error exception occurs, the processor will attempt to stack several words containing information about the state of the machine. If a bus error exception occurs during the stacking operation, there have been two bus errors in a row. This is commonly referred to as a double bus fault. When a double bus fault occurs, the processor will halt. Once a bus error exception has occurred, any bus error exception occurring before the execution of the next instruction constitutes a double bus fault.

Note that a bus cycle which is re-run does not constitute a bus error exception, and does not contribute to a double bus fault. Note also that this means that as long as the external hardware requests it, the processor will continue to re-run the same bus cycle.

The bus error pin also has an effect on processor operation after the processor receives an external reset input. As indicated in the section on reset operation (paragraph 4.2.4), the processor reads the vector table after a reset to determine the address to start program execution. If a bus error occurs while reading the vector table (or at any time before the first instruction is executed), the processor reacts as if a double bus fault has occurred and it halts. Only an external reset will start a halted processor.

4.2.4 RESET OPERATION. The reset signal is a bidirectional signal that allows either the processor or an external signal to reset the system. Figure 4-17 is a timing diagram for the reset operation. Both the halt and the reset lines must be applied to ensure total reset of the processor.

When the reset and halt lines are driven by an external device, it is recognized as an entire system reset, including the processor. The processor responds by reading the reset vector table entry (vector number zero, address $000000) and loads it into the supervisor stack pointer. Vector table entry number one at address $000004 is read next and loaded into the program counter. The processor initializes the status register to an interrupt level of seven. No other registers are affected by the reset sequence.

When a reset (RESET) instruction is executed, the processor drives the reset pin for 124 clock pulses. In this case, the processor is trying to reset the rest of the system. Therefore, there is no effect on the internal state of the processor. All of the processor's internal registers and the status register are unaffected by the execution of a reset instruction. All external devices connected to the reset line should be reset at the completion of the reset instruction.

When V_{CC} is initially applied to the processor, an external reset must be applied to the reset pin for 100 milliseconds.

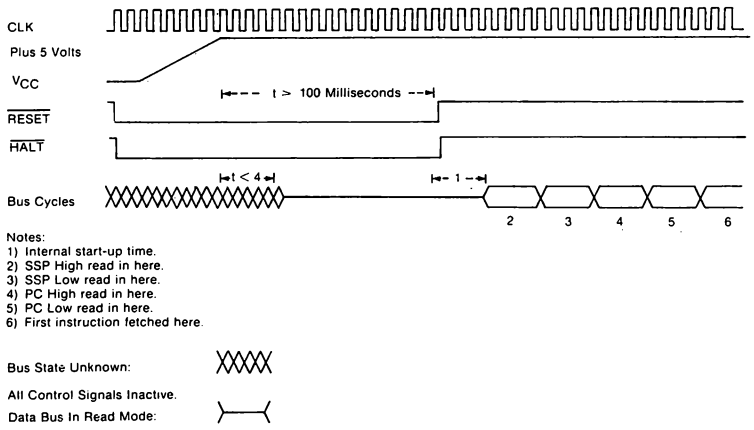


Figure 4-17. Reset Operation Timing Diagram

CHAPTER 5
EXCEPTION PROCESSING

5.0 INTRODUCTION

This chapter describes the actions of the MC68000 which are outside the normal processing associated with the execution of instructions. The functions of the bits in the supervisor portion of the status register are covered: the supervisor/user bit, the trace enable bit, and the processor priority mask. Finally, the sequence of memory references and actions taken by the processor on exception conditions is detailed.

5.1 PROCESSING STATES

The MC68000 is always in one of three processing states: normal, exception, or halted. The normal processing state is that associated with instruction execution; the memory references are to fetch instructions and operands, and to store results. A special case of the normal state is the stopped state which the processor enters when a STOP instruction is executed. In this state, no further memory references are made.

The exception processing state is associated with interrupts, trap instructions, tracing and other exceptional conditions. The exception may be internally generated by an instruction or by an unusual condition arising during the execution of an instruction. Externally, exception processing can be forced by an interrupt, by a bus error, or by a reset. Exception processing is designed to provide an efficient context swtich so that the processor may handle unusual conditions.

The halted processing state is an indication of catastrophic hardware failure. For example, if during the exception processing of a bus error another bus error occurs, the processor assumes that the system is unusable and halts. Only an external reset can restart a halted processor. Note that a processor in the stopped state is not in the halted state, nor vice versa.

5.2 PRIVILEGE STATES

The processor operates in one of two states of privilege: the "user" state or the "supervisor" state. The privilege state determines which operations are legal, is used by the external memory management device to control and translate accesses, and is used to choose between the supervisor stack pointer and the user stack pointer in instruction references.

The privilege state is a mechanism for providing security in a computer system. Programs should access only their own code and data areas, and ought to be restricted from accessing information which they do not need and must not modify.

The privilege mechanism provides security by allowing most programs to execute in user state. In this state, their accesses are controlled, and their effects on other parts of the system are limited. The operating system executes in the supervisor state, has access to all resources, and performs the overhead tasks for the user state programs.

5.2.1 SUPERVISOR STATE. The supervisor state is the higher state of privilege. For instruction execution, the supervisor state is determined by the S-bit of the status register; if the S-bit is asserted (high), the processor is in the supervisor state. All instructions can be executed in the supervisor state. The bus cycles generated by instructions executed in the supervisor state are classified as supervisor references. While the processor is in the supervisor privilege state, those instructions which use either the system stack pointer implicitly or address register seven explicitly access the supervisor stack pointer.

All exception processing is done in the supervisor state, regardless of the setting of the S-bit. The bus cycles generated during exception processing are classified as supervisor references. All stacking operations during exception processing use the supervisor stack pointer.

5.2.2 USER STATE. The user state is the lower state of privilege. For instruction execution, the user state is determined by the S-bit of the status register; if the S-bit is negated (low), the processor is executing instructions in the user state.

Most instructions execute the same in user state as in the supervisor state. However, some instructions which have important system effects are made privileged. User programs are not permitted to execute the STOP instruction, or the RESET instruction. To ensure that a user program cannot enter the supervisor state except in a controlled manner, the instructions which modify the whole status register are privileged. To aid in debugging programs which are to be used as operating systems, the move to user stack pointer (MOVE to USP) and move from user stack pointer (MOVE from USP) instructions are also privileged.

The bus cycles generated by an instruction executed in user state are classified as user state references. This allows an external memory management device to translate the address and to control access to protected portions of the address space. While the processor is in the user privilege state, those instructions which use either the system stack pointer implicitly, or address register seven explicitly, access the user stack pointer.

5.2.3 PRIVILEGE STATE CHANGES. Once the processor is in the user state and executing instructions, only exception processing can change the privilege state. During exception processing, the current setting of the S-bit of the status register is saved and the S-bit is asserted, putting the processing in the supervisor state. Therefore, when instruction execution resumes at the address specified to process the exception, the processor is in the supervisor privilege state.

The transition from supervisor to user state can be accomplished by any of four instructions: return from exception (RTE), move to status register (MOVE word to SR), AND immediate to status register (ANDI word to SR), and exclusive OR immediate to status register (EORI word to SR). The RTE instruction fetches the new status register and program counter from the supervisor stack, loads each into its respective register, and then begins the instruction fetch at the new program counter address in the privilege state determined by the S-bit of the new contents of the status register. The MOVE, ANDI, and EORI to Status Register instructions each fetch all operands in the supervisor state, perform the appropriate update to the status register, and then fetch the next instruction at the next sequential program counter address in the privilege state determined by the new S-bit.

5.2.4 REFERENCE CLASSIFICATION. When the processor makes a reference, it classifies the kind of reference being made, using the encoding on the three function code output lines. This allows external translation of addresses, control of access, and differentation of special processor states, such as interrupt acknowledge. Table 5-1 lists the classification of references.

Table 5-1. Reference Classification

Function Code Output			
FC2	FC1	FC0	Reference Class
0	0	0	(Unassigned)
0	0	1	User Data
0	1	0	User Program
0	1	1	(Unassigned)
1	0	0	(Unassigned)
1	0	1	Supervisor Data
1	1	0	Supervisor Program
1	1	1	Interrupt Acknowledge

If the four address spaces, user program, user data, supervisor program, and supervisor data, are externally separated, the processor can address up to 64 megabytes of memory.

5.3 EXCEPTION PROCESSING

Before discussing the details of interrupts, traps, and tracing, a general description of exception processing is in order. The processing of an exception occurs in four steps, with variations for different exception causes. During the first step, a temporary copy of the status register is made, and the status register is set for exception processing. In the second step the exception vector is determined, and the third step is the saving of the current processor context. In the fourth step a new context is obtained, and the processor switches to instruction processing.

5.3.1 EXCEPTION VECTORS. Exception vectors are memory locations from which the processor fetches the address of a routine which will handle that exception. All exception vectors are two words in length (figure 5-1), except for the reset vector, which is four words. All exception vectors lie in the supervisor data space, except for the reset vector which is in the supervisor program space. A vector number is an eight-bit number which, when multiplied by four, gives the address of an exception vector. Vector numbers are generated internally or externally, depending on the cause of the exception. In the case of interrupts, during the interrupt acknowledge bus cycle, a peripheral provides an 8-bit vector number (figure 5-2) to the processor on data bus lines D0 through D7. The processor translates the vector number into a full 24-bit address as shown in figure 5-3. The memory layout for exception vectors is given in table 5-2.

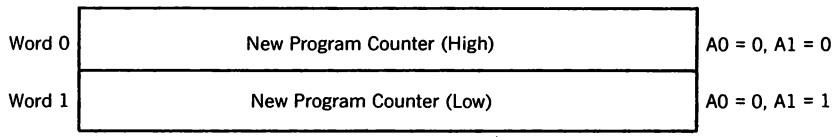
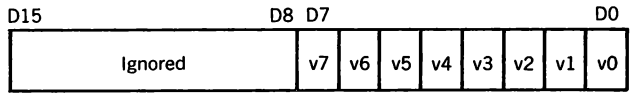


Figure 5-1. Exception Vector Format



Where:
v7 is the MSB of the Vector Number
v0 is the LSB of the Vector Number

Figure 5-2. Peripheral Vector Number Format

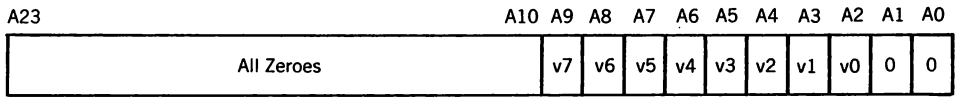


Figure 5-3. Address Translated from 8-Bit Vector Number

Table 5-2. Exception Vector Assignment

Vector Number(s)	Address			Assignment
	Dec	Hex	Space	
0	0	000	SP	Reset: Initial SSP
	4	004	SP	Reset: Initial PC
2	8	008	SD	Bus Error
3	12	00C	SD	Address Error
4	16	010	SD	Illegal Instruction
5	20	014	SD	Zero Divide
6	24	018	SD	CHK Instruction
7	28	01C	SD	TRAPV Instruction
8	32	020	SD	Privilege Violation
9	36	024	SD	Trace
10	40	028	SD	Line 1010 Emulator
11	44	02C	SD	Line 1111 Emulator
12*	48	030	SD	(Unassigned, reserved)
13*	52	034	SD	(Unassigned, reserved)
14*	56	038	SD	(Unassigned, reserved)
15*	60	03C	SD	(Unassigned, reserved)
16-23*	64	040	SD	(Unassigned, reserved)
	95	05F		—
24	96	060	SD	Spurious Interrupt
25	100	064	SD	Level 1 Interrupt Auto-Vector
26	104	068	SD	Level 2 Interrupt Auto-Vector
27	108	06C	SD	Level 3 Interrupt Auto-Vector
28	112	070	SD	Level 4 Interrupt Auto-Vector
29	116	074	SD	Level 5 Interrupt Auto-Vector
30	120	078	SD	Level 6 Interrupt Auto-Vector
31	124	07C	SD	Level 7 Interrupt Auto-Vector
32-47	128	080	SD	TRAP Instruction Vectors
	191	0BF		—
48-63*	192	0C0	SD	(Unassigned, reserved)
	255	0FF		—
64-255	256	100	SD	User Interrupt Vectors
	1023	3FF		—

*Vector numbers 12 through 23 and 48 through 63 are reserved for future enhancements by Motorola. No user peripheral devices should be assigned these numbers.

As shown in table 5-2, the memory layout is 512 words long (1024 bytes). It starts at address 0 and proceeds through address 1023. This provides 255 unique vectors; some of these are reserved for TRAPS and other system functions. Of the 255, there are 192 reserved for user interrupt vectors. However, there is no protection on the first 64 entries, so user interrupt vectors may overlap at the discretion of the systems designer.

5.3.2 KINDS OF EXCEPTIONS. Exceptions can be generated by either internal or external causes. The externally generated exceptions are the interrupts and the bus error and reset requests. The interrupts are requests from peripheral devices for processor action while the bus error and reset inputs are used for access control and processor restart. The internally generated exceptions come from instructions, or from address errors or tracing. The trap (TRAP), trap on overflow (TRAPV), check register against bounds (CHK) and divide (DIV) instructions all can generate exceptions as part of their instruction execution. In addition, illegal instructions, word fetches from odd addresses and privilege violations cause exceptions. Tracing behaves like a very high priority, internally generated interrupt after each instruction execution.

5.3.3 EXCEPTION PROCESSING SEQUENCE. Exception processing occurs in four identifiable steps. In the first step, an internal copy is made of the status register. After the copy is made, the S-bit is asserted, putting the processor into the supervisor privilege state. Also, the T-bit is negated which will allow the exception handler to execute unhindered by tracing. For the reset and interrupt exceptions, the interrupt priority mask is also updated.

In the second step, the vector number of the exception is determined. For interrupts, the vector number is obtained by a processor fetch, classified as an interrupt acknowledge. For all other exceptions, internal logic provides the vector number. This vector number is then used to generate the address of the exception vector.

The third step is to save the current processor status, except for the reset exception. The current program counter value and the saved copy of the status register are stacked using the supervisor stack pointer. The program counter value stacked usually points to the next unexecuted instruction, however for bus error and address error, the value stacked for the program counter is unpredictable, and may be incremented from the address of the instruction which caused the error. Additional information defining the current context is stacked for the bus error and address error exceptions.

The last step is the same for all exceptions. The new program counter value is fetched from the exception vector. The processor then resumes instruction execution. The instruction at the address given in the exception vector is fetched, and normal instruction decoding and execution is started.

5.3.4 MULTIPLE EXCEPTIONS. These paragraphs describe the processing which occurs when multiple exceptions arise simultaneously. Exceptions can be grouped according to their occurrence and priority. The Group 0 exceptions are reset, bus error, and address error. These exceptions cause the instruction currently being executed to be aborted, and the exception processing to commence at the next minor cycle of the processor. The Group 1 exceptions are trace and interrupt, as well as the privilege violations and illegal instructions. These exceptions allow the current

instruction to execute to completion, but preempt the execution of the next instruction by forcing exception processing to occur (privilege violations and illegal instructions are detected when they are the next instruction to be executed). The Group 2 exceptions occur as part of the normal processing of instructions. The TRAP, TRAPV, CHK, and zero divide exceptions are in this group. For these exceptions, the normal execution of an instruction may lead to exception processing.
Group 0 exceptions have highest priority, while Group 2 exceptions have lowest priority. Within Group 0, reset has highest priority, followed by bus error and then address error. Within Group 1, trace has priority over external interrupts, which in turn takes priority over illegal instruction and privilege violation. Since only one instruction can be executed at a time, there is no priority relation within Group 2.

The priority relation between two exceptions determines which is taken, or taken first, if the conditions for both arise simultaneously. Therefore, if a bus error occurs during a TRAP instruction, the bus error takes precedence, and the TRAP instruction processing is aborted. In another example, if an interrupt request occurs during the execution of an instruction while the T-bit is asserted, the trace exception has priority, and is processed first. Before instruction execution resumes however, the interrupt exception is also processed, and instruction processing commences finally in the interrupt handler routine. A summary of exception grouping and priority is given in table 5-3.

Table 5-3. Exception Grouping and Priority

Group	Exception	Processing
0	Reset Bus Error Address Error	Exception processing begins at the next minor cycle
1	Trace Interrupt Illegal Privilege	Exception processing begins before the next instruction
2	TRAP, TRAPV, CHK, Zero Divide	Exception processing is started by normal instruction execution

5.4 EXCEPTION PROCESSING DETAILED DISCUSSION

Exceptions have a number of sources, and each exception has processing which is peculiar to it. The following paragraphs detail the sources of exceptions, how each arises, and how each is processed.

5.4.1 RESET. The reset input provides the highest level exception. The processing of the reset signal is designed for system initiation, and recovery from catastrophic failure. Any processing in progress at the time of the reset is aborted and cannot be recovered. The processor is forced into the supervisor state, and the trace state is

forced off. The processor interrupt priority mask is set at level seven. The vector number is internally generated to reference the reset exception vector at location 0 in the supervisor program space. Because no assumptions can be made about the validity of register contents, in particular the supervisor stack pointer, neither the program counter nor the status register is saved. The address contained in the first two words of the reset exception vector is fetched as the initial supervisor stack pointer, and the address in the last two words of the reset exception vector is fetched as the initial program counter. Finally, instruction execution is started at the address in the program counter. The power-up/restart code should be pointed to by the initial program counter.

The RESET instruction does not cause loading of the reset vector, but does assert the reset line to reset external devices. This allows the software to reset the system to a known state and then continue processing with the next instruction.

5.4.2 INTERRUPTS. Seven levels of interrupt priorities are provided. Devices may be chained externally within interrupt priority levels, allowing an unlimited number of peripheral devices to interrupt the processor. Interrupt priority levels are numbered from one to seven, level seven being the highest priority. The status register contains a three-bit mask which indicates the current processor priority, and interrupts are inhibited for all priority levels less than or equal to the current processor priority.

An interrupt request is made to the processor by encoding the interrupt request level on the interrupt request lines; a zero indicates no interrupt request. Interrupt requests arriving at the processor do not force immediate exception processing, but are made pending. Pending interrupts are detected between instruction executions. If the priority of the pending interrupt is lower than or equal to the current processor priority, execution continues with the next instruction and the interrupt exception processing is postponed. (The recognition of level seven is slightly different, as explained in the following paragraph).

If the priority of the pending interrupt is greater than the current processor priority, the exception processing sequence is started. First a copy of the status register is saved, and the privilege state is set to supervisor, tracing is suppressed, and the processor priority level is set to the level of the interrupt being acknowledged. The processor fetches the vector number from the interrupting device, classifying the reference as an interrupt acknowledge and displaying the level number of the interrupt being acknowledged on the address bus. If external logic requests an automatic vectoring, the processor internally generates a vector number which is determined by the interrupt level number (see para. 6.2). If external logic indicates a bus error, the interrupt is taken to be spurious, and the generated vector number references the spurious interrupt vector. The processor then proceeds with the usual exception processing, saving the program counter and status register on the supervisor stack. The saved value of the program counter is the address of the instruction which would

have been executed had the interrupt not been present. The content of the interrupt vector whose vector number was previously obtained is fetched and loaded into the program counter, and normal instruction execution commences in the interrupt handling routine. A flow chart for the interrupt acknowledge sequence is given in figure 5-4; a timing diagram is given in figure 5-5.

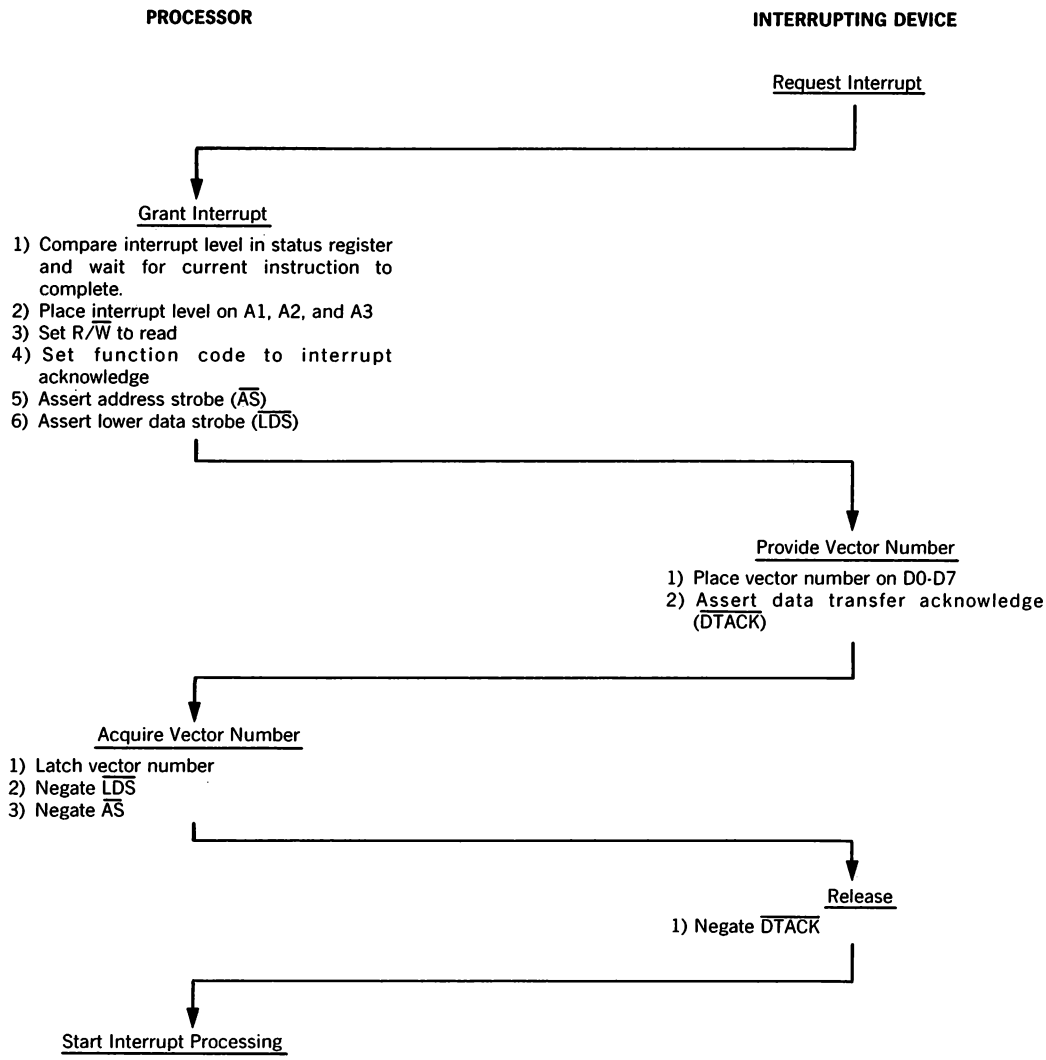

PROCESSOR
INTERRUPTING DEVICE
Request Interrupt
Grant Interrupt
1) Compare interrupt level in status register and wait for current instruction to complete.
2) Place interrupt level on A1, A2, and A3
3) Set R/W to read
4) Set function code to interrupt acknowledge
5) Assert address strobe (AS)
6) Assert lower data strobe (LDS)
Provide Vector Number
1) Place vector number on D0-D7
2) Assert data transfer acknowledge (DTACK)
Acquire Vector Number
1) Latch vector number
2) Negate LDS
3) Negate AS
Release
1) Negate DTACK
Start Interrupt Processing

Figure 5-4. Interrupt Acknowledge Sequence Flow Chart

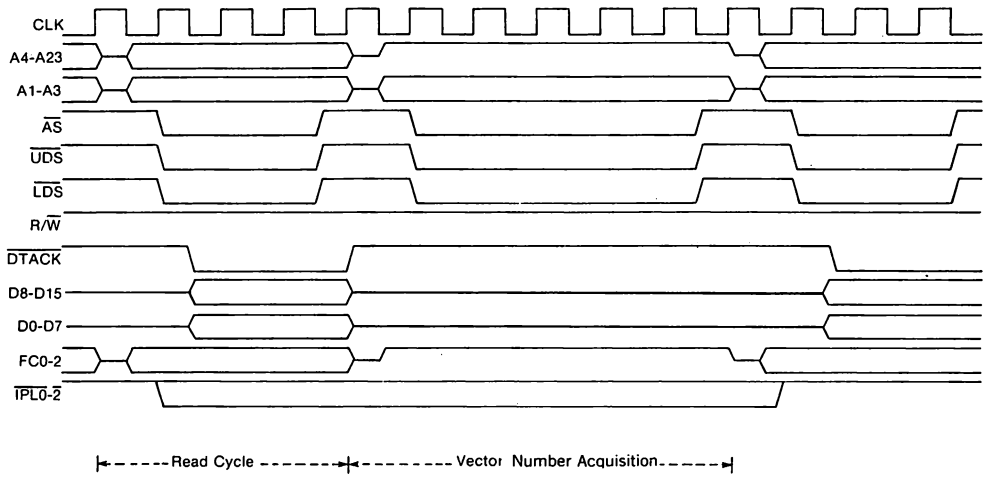


Figure 5-5. Interrupt Acknowledge Sequence Timing Diagram

Priority level seven is a special case. Level seven interrupts cannot be inhibited by the interrupt priority mask, thus providing a "non-maskable interrupt" capability. An interrupt is generated each time the interrupt request level changes from some lower level to level seven. Note that a level seven interrupt may still be caused by the level comparison if the request level is a seven and the processor priority is set to a lower level by an instruction.

5.4.3 INSTRUCTION TRAPS. Traps are exceptions caused by instructions. They arise either from processor recognition of abnormal conditions during instruction execution, or from use of instructions whose normal behavior is trapping.

Exception processing for traps is straightforward. The status register is copied, the supervisor state is entered, and the trace state is turned off. The vector number is internally generated; for the TRAP instruction, part of the vector number comes from the instruction itself. The program counter and the copy of the status register are saved on the supervisor stack. The saved value of the program counter is the address of the instruction after the instruction which generated the trap. Finally, instruction execution commences at the address contained in the exception vector.

Some instructions are used specifically to generate traps. The TRAP instruction always forces an exception, and is useful for implementing system calls for user programs. The TRAPV and CHK instructions force an exception if the user program detects a runtime error, which may be an arithmetic overflow or a subscript out of bounds.

The signed divide (DIVS) and unsigned divide (DIVU) instructions will force an exception if a division operation is attempted with a divisor of zero.

5.4.4 ILLEGAL AND UNIMPLEMENTED INSTRUCTIONS. Illegal instruction is the term used to refer to any of the word bit patterns which are not the bit pattern of the first word of a legal instruction. During instruction execution, if such an instruction is fetched, an illegal instruction exception occurs.

Word patterns with bits 15 through 12 equaling 1010 or 1111 are distinguished as unimplemented instructions and separate exception vectors are given to these patterns to permit efficient emulation. This facility allows the operating system to detect program errors, or to emulate unimplemented instructions in software.

Exception processing for illegal instructions is similar to that for traps. After the instruction is fetched and decoding is attempted, the processor determines that execution of an illegal instruction is being attempted and starts exception processing. The status register is copied, the supervisor state is entered, and the trace state is turned off. The vector number is generated to refer to the illegal instruction vector, or in the case of unimplemented instructions, to the corresponding emulation vector. The current program counter and copy of the status register are saved on the supervisor stack, with the saved value of the program counter being the address of the illegal instruction. Finally, instruction commences at the address contained in the exception vector.

5.4.5 PRIVILEGE VIOLATIONS. In order to provide system security, various instructions are privileged. An attempt to execute one of the privileged instructions while in the user state will cause an exception. The privileged instructions are:

STOP
RESET
RTE
MOVE to SR
AND (word) Immediate to SR
EOR (word) Immediate to SR
OR (word) Immediate to SR
MOVE USP

Exception processing for privilege violations is nearly identical to that for illegal instructions. After the instruction is fetched and decoded, and the processor determines that a privilege violation is being attempted, the processor starts exception processing. The status register is copied, the supervisor state is entered, and the trace state is turned off. The vector number is generated to reference the privilege violation vector, and the current program counter and the copy of the status register are saved on the supervisor stack. The saved value of the program counter is the address of the first word of the instruction which caused the privilege violation. Finally, instruction execution commences at the address contained in the privilege violation exception vector.

5.4.6 TRACING. To aid in program development, the MC68000 includes a facility to allow instruction by instruction tracing. In the trace state, after each instruction is executed an exception is forced, allowing a debugging program to monitor the execution of the program under test.

The trace facility uses the T-bit in the supervisor portion of the status register. If the T-bit is negated (off), tracing is disabled, and instruction execution proceeds from instruction to instruction as normal. If the T-bit is asserted (on) at the beginning of the execution of an instruction, a trace exception will be generated after the execution of that instruction is completed. If the instruction is not executed, either because an interrupt is taken, or the instruction is illegal or privileged, the trace exception does not occur. The trace exception also does not occur if the instruction is aborted by a reset, bus error, or address error exception. If the instruction is indeed executed and an interrupt is pending on completion, the trace exception is processed before the interrupt exception. If, during the execution of the instruction, an exception is forced by that instruction, the forced exception is processed before the trace exception.

As an extreme illustration of the above rules, consider the arrival of an interrupt during the execution of a TRAP instruction while tracing is enabled. First the trap exception is processed, then the trace exception, and finally the interrupt exception. Instruction execution resumes in the interrupt handler routine.

The exception processing for trace is quite simple. After the execution of the instruction is completed, and before the start of the next instruction, exception processing begins. A copy is made of the status register. The transition to supervisor privilege state is made, and as usual, the T-bit of the status register is turned off, disabling further tracing. The vector number is generated to reference the trace exception vector, and the current program counter and the copy of the status register are saved on the supervisor stack. The saved value of the program counter is the address of the next instruction. Instruction execution commences at the address contained in the trace exception vector.

5.4.7 BUS ERROR. Bus error exceptions occur when external logic requests that a bus error be processed by an exception. The current bus cycle which the processor is making is aborted. Whether the processor was doing instruction or exception processing, that processing is terminated, and the processor immediately begins exception processing.

Exception processing for bus error follows the usual sequence of steps. The status register is copied, the supervisor state is entered, and the trace state is turned off. The vector number is generated to refer to the bus error vector. Since the processor was not between instructions when the bus error exception request was made, the context of the processor is more detailed. To save more of this context, additional information is saved on the supervisor stack. The program counter and the copy of the status register are of course saved. The value saved for the program counter is advanced by some amount, two to ten bytes beyond the address of the first word of the instruction

which made the reference causing the bus error. If the bus error occurred during the fetch of the next instruction, the saved program counter has a value in the vicinity of the current instruction, even if the current instruction is a branch, a jump or a return instruction. Besides the usual information, the processor saves its internal copy of the first word of the instruction being processed, and the address which was being accessed by the aborted bus cycle. Specific information about the access is also saved whether it was a read or a write, whether the processor was processing an instruction or not, and the classification displayed on the function code outputs when the bus error occurred. The processor is processing an instruction if it is in the normal state or processing a Group 2 exception; the processor is not processing an instruction if it is processing a Group 0 or a Group 1 exception. Figure 5-6 illustrates how this information is organized on the supervisor stack. Although this information is not sufficient in general to effect full recovery from the bus error, it does allow software diagnosis. Finally, the processor commences instruction processing at the address contained in the vector. It is the responsibility of the error handler routine to clean up the stack and determine where to continue execution.

If a bus error occurs during the exception processing for a bus error, address error, or reset, the processor is halted, and all processing ceases. This simplifies the detection of catastrophic system failure, since the processor removes itself from the system rather than destroying all memory contents. Only the RESET pin can restart a halted processor.

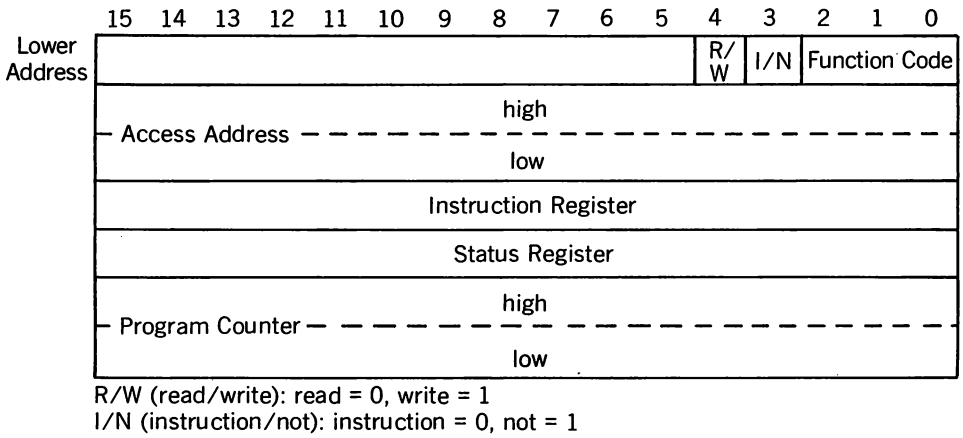


Figure 5-6. Supervisor Stack Order for Bus or Address Error Exception

5.4.8 ADDRESS ERROR. Address error exceptions occur when the processor attempts to access a word or a long word operand or an instruction at an odd address. The effect is much like an internally generated bus error, so that the bus cycle is aborted, and the processor ceases whatever processing it is currently doing and begins exception processing. After exception processing commences, the sequence is the same as that for bus error including the information that is stacked, except that the vector number refers to the address error vector instead. Likewise, if an address error occurs during the exception processing for a bus error, address error, or reset, the processor is halted.

CHAPTER 6
INTERFACE WITH M6800 PERIPHERALS

6.0 INTRODUCTION

Motorola's extensive line of M6800 peripherals are directly compatible with the MC68000. Some of these devices that are particularly useful are:

MC6821 Peripheral Interface Adapter
MC6840 Programmable Timer Module
MC6843 Floppy Disk Controller
MC6845 CRT Controller
MC6849 Dual Density Floppy Disk Controller
MC6850 Asynchronous Communication Interface Adapter
MC6852 Synchronous Serial Data Adapter
MC6854 Advanced Data Link Controller
MC68488 General Purpose Interface Adapter

To interface the synchronous M6800 peripherals with the asynchronous MC68000, the processor modifies its bus cycle to meet the M6800 cycle requirements whenever an M6800 device address is detected. This is possible since both processors use memory mapped I/O. Figure 6-1 is a flow chart of the interface operation between the processor and M6800 devices.

6.1 DATA TRANSFER OPERATION

Three signals on the processor provide the M6800 interface. They are: enable (E), valid memory address ($\overline{VMA}$), and valid peripheral address ($\overline{VPA}$). Figure 6-2 shows the timing required by M6800 peripherals, the timing specified for the M6800, and the corresponding timing for the MC68000. For further details on peripheral timing, consult The Complete Motorola Microcomputer Data Library. Notice that the MC68000 $\overline{VMA}$ is active low, contrasted with the active high M6800 VMA. This allows the processor to put its buses in the high impedance state on DMA requests without inadvertently selecting peripherals.

Enable corresponds to the E or $\phi 2$ signal in existing M6800 systems. It is the bus clock used by the M6800 peripherals to synchronize data transfer. Enable is a constant frequency clock that is one tenth of the incoming MC68000 clock frequency. The timing of E allows 1 MHz peripherals to be used with an 8 MHz MC68000. Enable has a 60/40 duty cycle; that is, it is low for six input clocks and high for four input clocks. This duty cycle allows the processor to do two successive $\overline{VPA}$ accesses on successive E pulses.

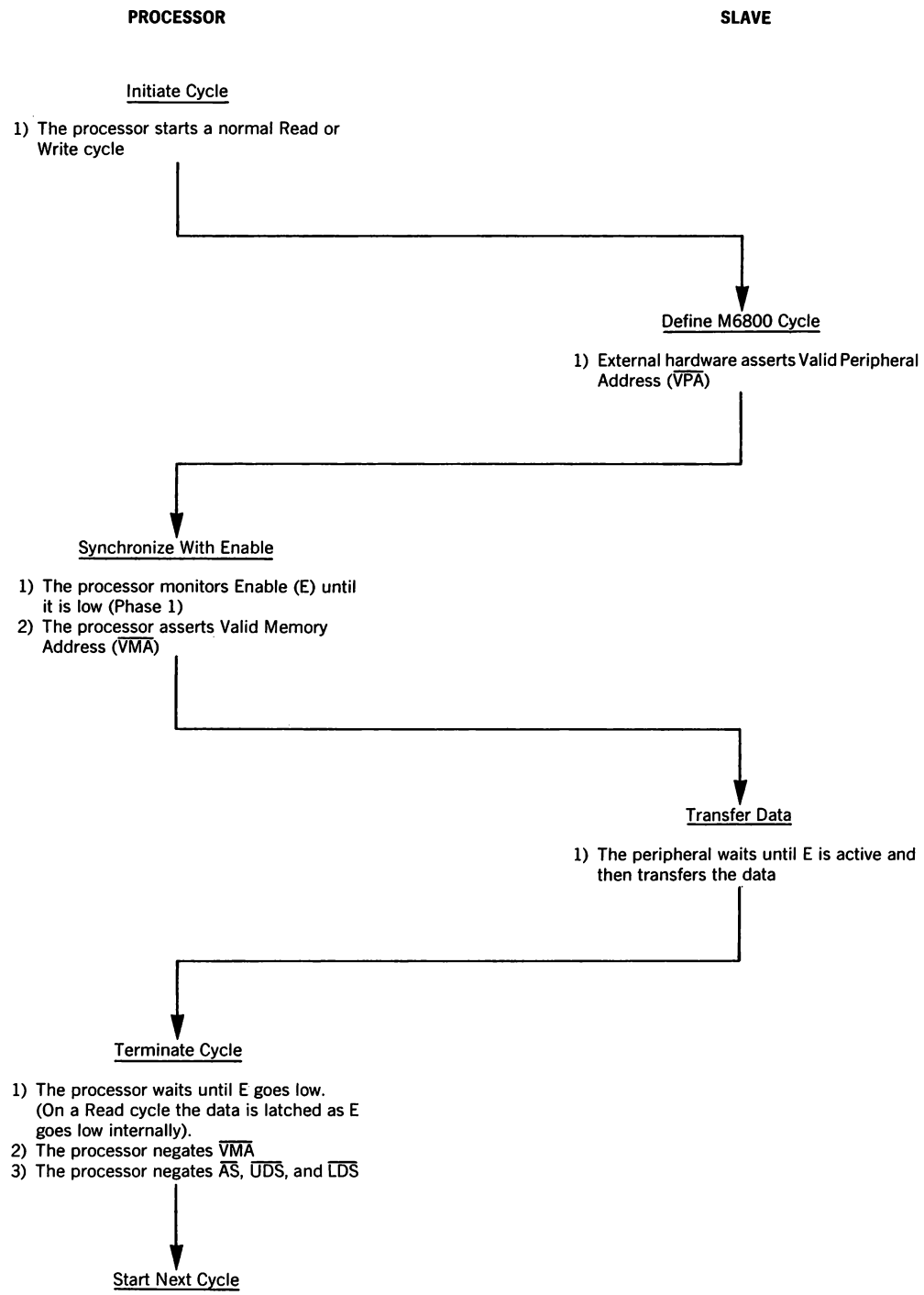
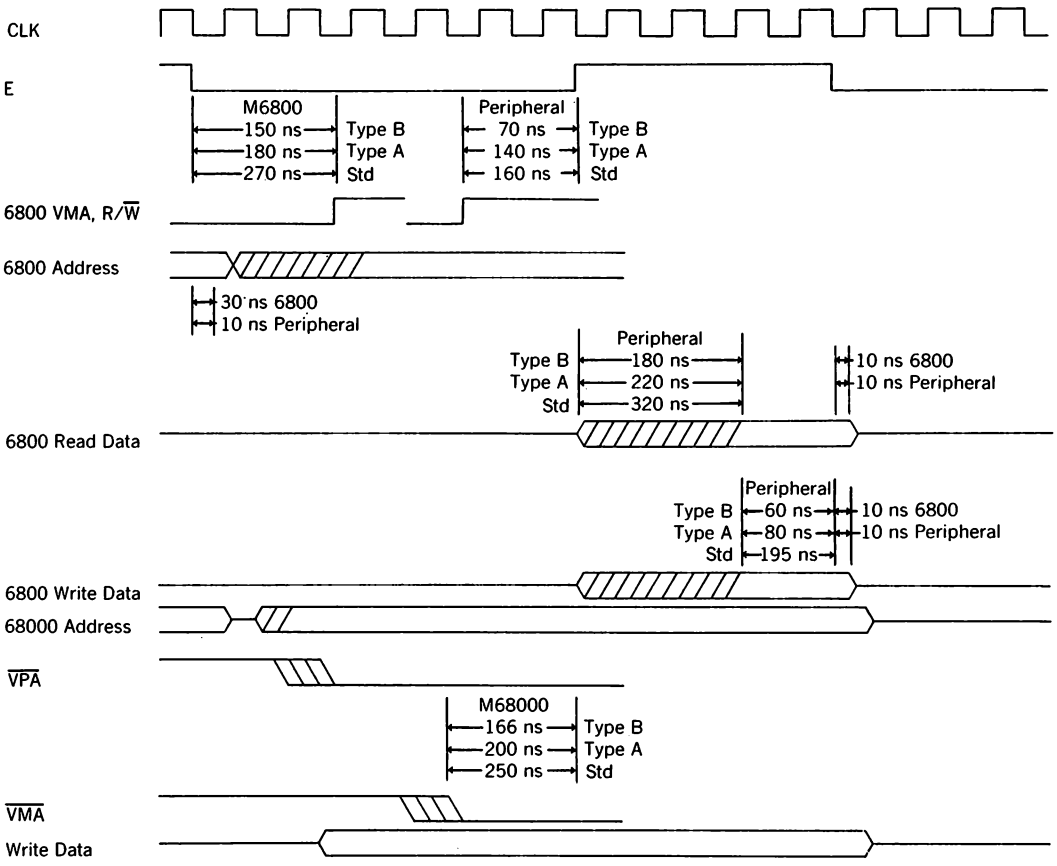

PROCESSOR
SLAVE
Initiate Cycle
1) The processor starts a normal Read or Write cycle
Define M6800 Cycle
1) External hardware asserts Valid Peripheral Address (VPA)
Synchronize With Enable
1) The processor monitors Enable (E) until it is low (Phase 1)
2) The processor asserts Valid Memory Address (VMA)
Transfer Data
1) The peripheral waits until E is active and then transfers the data
Terminate Cycle
1) The processor waits until E goes low. (On a Read cycle the data is latched as E goes low internally).
2) The processor negates VMA
3) The processor negates AS, UDS, and LDS
Start Next Cycle

Figure 6-1. M6800 Interfacing Flow Chart



NOTE:
Times are expressed for different device clock frequencies.

CLK. FREQ.	F-TYPE
1.0 MHz	B
1.5 MHz	A
2.0 MHz	Std

SURPLUS TRAVEL TIME

F-TYPE	6800	68000
B	30 ns	96 ns
A	14 ns	60 ns
Std	70 ns	90 ns

Figure 6-2. MC68000 to M6800 Peripheral Timing Diagram

The $\overline{VPA}$ input signals the processor that the address on the bus is the address of an M6800 device (or an area reserved for M6800 devices) and that the bus should conform to the $\phi 2$ transfer characteristics of the M6800 bus. Valid peripheral address is derived by decoding the address bus, conditioned by address strobe.

As a result of receiving $\overline{VPA}$, the processor issues $\overline{VMA}$ with the appropriate timing when E is low. Valid memory address is then used as part of the chip select equation of the peripheral. This ensures that the M6800 peripherals are selected and deselected at the correct time.

As is shown, the setup and hold times provided by the MC68000 are better than required. Surplus travel time is the delay that can be tolerated in the $\overline{VMA}$ line. It is defined as the difference between the $\overline{VMA}$ setup provided by the processor and that required by the peripheral.

Two example systems with M6800 peripherals are shown in figures 6-3 and 6-4. The system in figure 6-3 reserves the upper eight megabytes of memory for M6800 peripherals. The system in figure 6-4 is more efficient with memory and easily expandable, but more complex.

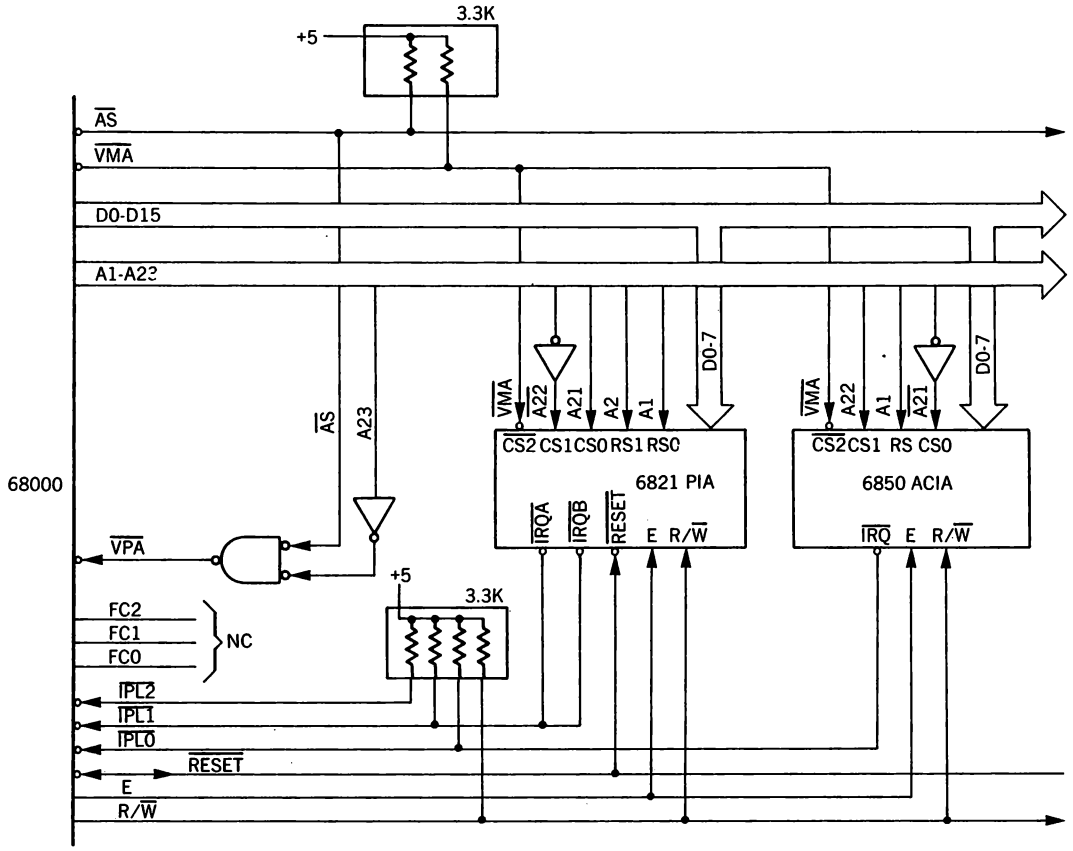


Figure 6-3. M6800 Interface — Example 1

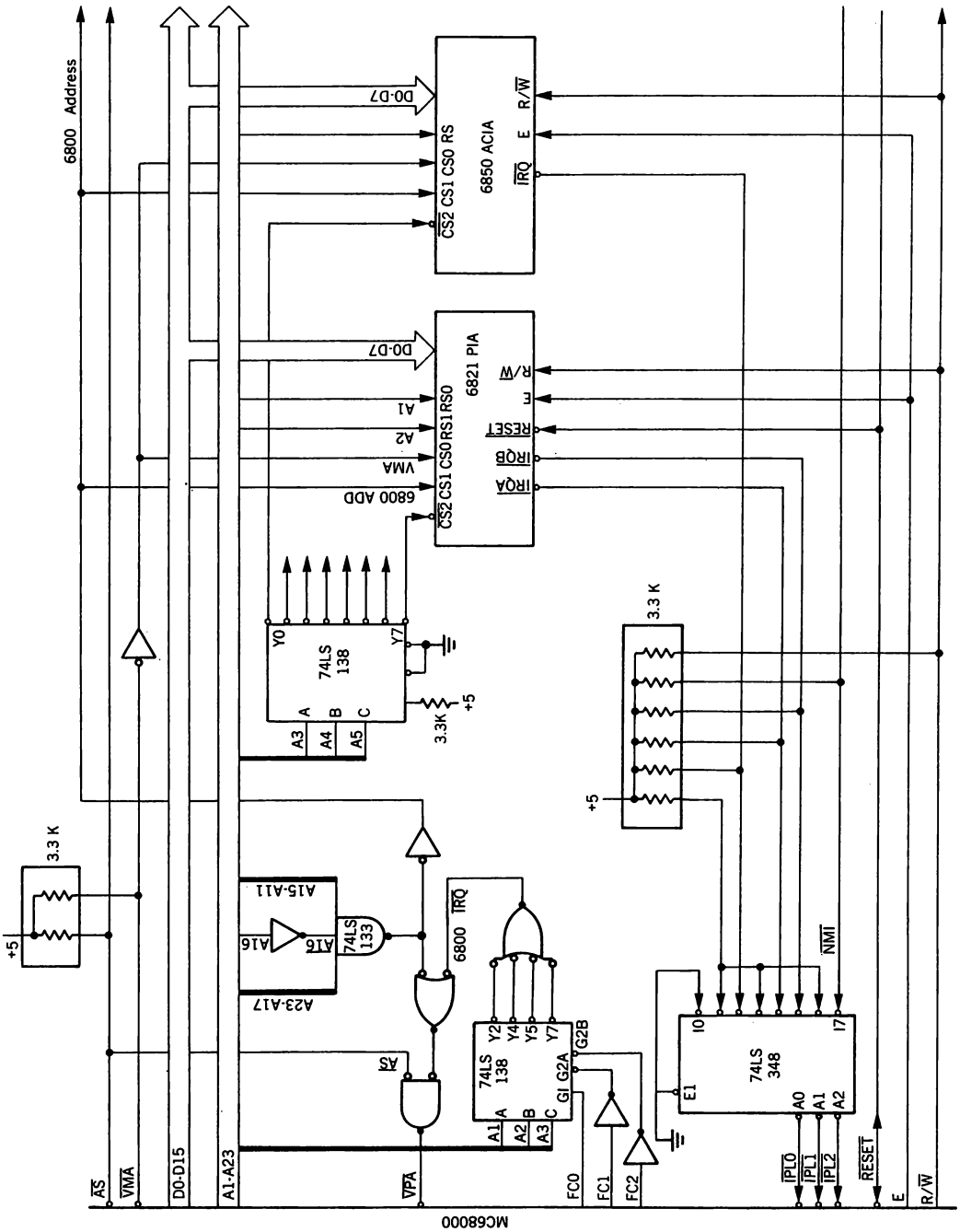


Figure 6-4. M6800 Interface — Example 2

6.2 INTERRUPT INTERFACE OPERATION

During an interrupt acknowledge cycle while the processor is fetching the vector, if $\overline{VPA}$ is asserted, the MC68000 will assert $\overline{VMA}$ and complete a normal M6800 read cycle, as shown in figure 6-5. The processor will then use an internally generated vector that is a function of the interrupt being serviced. This process is known as autovectoring. The seven autovectors are vector numbers 25 through 31 (decimal).

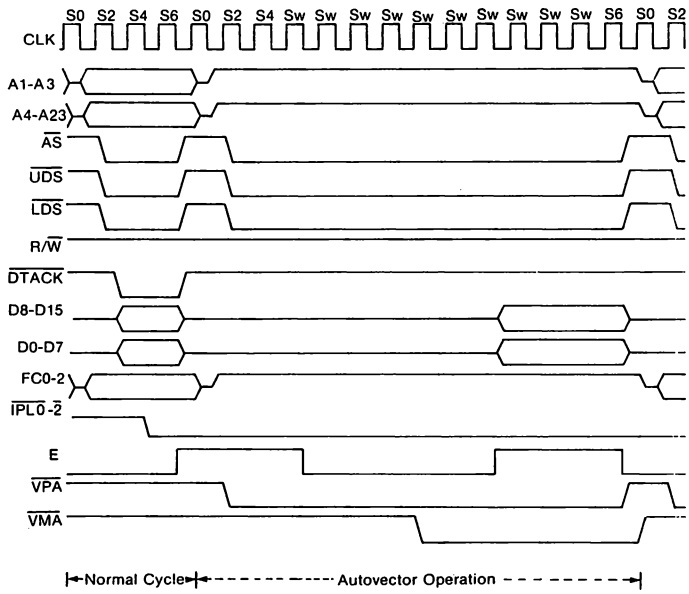


Figure 6-5. Autovector Operation Timing Diagram

This operates in the same fashion (but is not restricted to) the M6800 interrupt sequence. The basic difference is that there are six normal interrupt vectors and one NMI type vector. As with both the M6800 and the MC68000's normal vectored interrupt, the interrupt service routine can be located anywhere in the address space. This is due to the fact that while the vector numbers are fixed, the contents of the vector table entries are assigned by the user.

Since $\overline{VMA}$ is asserted during autovectoring, the M6800 peripheral address decoding should prevent unintended accesses. This is demonstrated in both examples. The system in figure 6-3 relies on the fact that when the MC68000 is fetching a vector, the upper bits of the address bus are ones to generate the $\overline{VPA}$ for autovectoring. Notice that in this system only autovector interrupts can be performed. Figure 6-4 shows a more flexible design.

APPENDIX A
CONDITION CODES COMPUTATION

A.0 INTRODUCTION

This appendix provides a discussion of how the condition codes were developed, the meanings of each bit, how they are computed, and how they are represented in the instruction set details.

Two criteria were used in developing the condition codes:

Consistency — across instruction, uses, and instances

Meaningful results — no change unless it provides useful information

The consistency across instructions means that instructions which are special cases of more general instructions affect the condition codes in the same way. Consistency across instances means that if an instruction ever affects a condition code, it will always affect that condition code. Consistency across uses means that whether the condition codes were set by a compare, test, or move instruction, the conditional instructions test the same situation. The tests used for the conditional instructions and the code computations are given in paragraph A.4.

A.1 CONDITION CODE REGISTER.

The condition code register portion of the status register contains five bits:

N — Negative

Z — Zero

V — Overflow

C — Carry

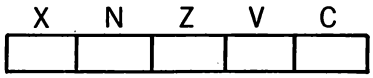
X — Extend

The first four bits are true condition code bits in that they reflect the condition of the result of a processor operation. The X-bit is an operand for multiprecision computations. The carry bit (C) and the multiprecision operand extend bit (X) are separate in the MC68000 to simplify the programming model.

A.2 CONDITION CODE REGISTER NOTATION

In the instruction set details given in Appendix B, the description of the effect on the condition codes is given in the following form:

Condition Codes:



where:

N (negative)	set if the most significant bit of the result is set. Cleared otherwise.
Z (zero)	set if the result equals zero. Cleared otherwise.
V (overflow)	set if there was an arithmetic overflow. This implies that the result is not representable in the operand size. Cleared otherwise.
C (carry)	set if a carry is generated out of the most significant bit of the operands for an addition. Also set if a borrow is generated in a subtraction. Cleared otherwise.
X (extend)	transparent to data movement. When affected, it is set the same as the C-bit.

The notational convention that appears in the representation of the condition code register is:

*	set according to the result of the operation
—	not affected by the operation
0	cleared
1	set
U	undefined after the operation

A.3 CONDITION CODE COMPUTATION

Most operations take a source operand and a destination operand, compute, and store the result in the destination location. Unary operations take a destination operand, compute, and store the result in the destination location. Table A-1 details how each instruction sets the condition codes.

Table A-1. Condition Code Computations

Operations	X	N	Z	V	C	Special Definition
ABCD	*	U	?	U	?	C = Decimal Carry $Z = Z \cdot \overline{Rm} \cdot \ldots \cdot \overline{R0}$
ADD, ADDI, ADDQ	*	*	*	?	?	$V = Sm \cdot Dm \cdot \overline{Rm} + \overline{Sm} \cdot \overline{Dm} \cdot Rm$ $C = Sm \cdot Dm + \overline{Rm} \cdot Dm + Sm \cdot \overline{Rm}$
ADDX	*	*	?	?	?	$V = Sm \cdot Dm \cdot \overline{Rm} + \overline{Sm} \cdot \overline{Dm} \cdot \overline{Rm}$ $C = Sm \cdot Dm + \overline{Rm} \cdot Dm + Sm \cdot \overline{Rm}$ $Z = Z \cdot \overline{Rm} \cdot \ldots \cdot \overline{R0}$
AND, ANDI, EOR, EORI, MOVEQ, MOVE, OR, ORI, CLR, EXT NOT, TAS, TST	—	*	*	0	0	
CHK	—	*	U	U	U	
SUB, SUBI SUBQ	*	*	*	?	?	$V = \overline{Sm} \cdot Dm \cdot \overline{Rm} + Sm \cdot \overline{Dm} \cdot Rm$ $C = Sm \cdot \overline{Dm} + Rm \cdot \overline{Dm} + Sm \cdot Rm$
SUBX	*	*	?	?	?	$V = \overline{Sm} \cdot Dm \cdot \overline{Rm} + Sm \cdot \overline{Dm} \cdot Rm$ $C = Sm \cdot \overline{Dm} + Rm \cdot \overline{Dm} + Sm \cdot Rm$ $Z = Z \cdot \overline{Rm} \cdot \ldots \cdot \overline{R0}$
CMP, CMPI, CMPM	—	*	*	?	?	$V = \overline{Sm} \cdot Dm \cdot \overline{Rm} + Sm \cdot \overline{Dm} \cdot Rm$ $C = Sm \cdot \overline{Dm} + Rm \cdot \overline{Dm} + Sm \cdot Rm$
DIVS, DIVU	—	*	*	?	0	V = Division Overflow
MULS, MULU	—	*	*	0	0	
SBCD, NBCD	*	U	?	U	?	C = Decimal Borrow $Z = Z \cdot \overline{Rm} \cdot \ldots \cdot \overline{R0}$
NEG	*	*	*	?	?	V = Dm · Rm, C = Dm + Rm
NEGX	*	*	?	?	?	V = Dm · Rm, C = Dm + Rm $Z = Z \cdot \overline{Rm} \cdot \ldots \cdot \overline{R0}$
BTST, BCHG, BSET, BCLR	—	—	?	—	—	$Z = \overline{Dn}$
ASL	*	*	*	?	?	$V = Dm \cdot (\overline{D_{m-1}} + \ldots + \overline{D_{m-r}}) + \overline{Dm} \cdot (D_{m-1} + \ldots + D_{m-r})$ $C = D_{m-r+1}$
ASL (r = 0)	—	*	*	0	0	
LSL, ROXL	*	*	*	0	?	$C = D_{m-r+1}$
LSR (r = 0)	—	*	*	0	0	
ROXL (r = 0)	—	*	*	0	?	C = X
ROL	—	*	*	0	?	$C = D_{m-r+1}$
ROL (r = 0)	—	*	*	0	0	
ASR, LSR, ROXR	*	*	*	0	?	$C = D_{r-1}$
ASR, LSR (r = 0)	—	*	*	0	0	
ROXR (r = 0)	—	*	*	0	?	C = X
ROR	—	*	*	0	?	$C = D_{r-1}$
ROR (r = 0)	—	*	*	0	0	

— Not affected
U Undefined
? Other — see Special Definition

* General Case:
X = C
N = Rm
$Z = \overline{Rm} \cdot \ldots \cdot \overline{R0}$

Sm — Source operand most significant bit
Dm — Destination operand most significant bit
Rm — Result bit most significant bit
n — bit number
r — shift amount

A.4 CONDITIONAL TESTS

Table A-2 lists the condition names, encodings, and tests for the conditional branch and set instructions. The test associated with each condition is a logical formula based on the current state of the condition codes. If this formula evaluates to 1, the condition succeeds, or is true. If the formula evaluates to 0, the condition is unsuccessful, or false. For example, the T condition always succeeds , while the EQ condition succeeds only if the Z bit is currently set in the condition codes.

Table A-2. Conditional Tests

Mnemonic	Condition	Encoding	Test
T	true	0000	1
F	false	0001	0
HI	high	0010	$\overline{C} \cdot \overline{Z}$
LS	low or same	0011	$C + Z$
CC	carry clear	0100	$\overline{C}$
CS	carry set	0101	C
NE	not equal	0110	$\overline{Z}$
EQ	equal	0111	Z
VC	overflow clear	1000	$\overline{V}$
VS	overflow set	1001	V
PL	plus	1010	$\overline{N}$
MI	minus	1011	N
GE	greater or equal	1100	$N \cdot V + \overline{N} \cdot \overline{V}$
LT	less than	1101	$N \cdot \overline{V} + \overline{N} \cdot V$
GT	greater than	1110	$N \cdot V \cdot \overline{Z} + \overline{N} \cdot \overline{V} \cdot \overline{Z}$
LE	less or equal	1111	$Z + N \cdot \overline{V} + \overline{N} \cdot V$

APPENDIX B
INSTRUCTION SET DETAILS

B.0 INTRODUCTION

This appendix contains detailed information about each instruction in the MC68000 instruction set. They are arranged in alphabetical order with the mnemonic heading set in large bold type for easy reference.

B.1 ADDRESSING CATEGORIES

Effective address modes may be categorized by the ways in which they may be used. The following classifications will be used in the instruction definitions.

Data — If an effective address mode may be used to refer to data operands, it is considered a data addressing effective address mode.

Memory — If an effective address mode may be used to refer to memory operands, it is considered a memory addressing effective address mode.

Alterable — If an effective address mode may be used to refer to alterable (writable) operands, it is considered an alterable addressing effective address mode.

Control — If an effective address mode may be used to refer to memory operands without an associated size, it is considered a control addressing effective address mode.

Table B-1 shows the various categories to which each of the effective address modes belongs.

Table B-1. Effective Addressing Mode Categories

Effective Address Modes	Mode	Register	Addressing Categories				Assembler Syntax
			Data	Memory	Control	Alterable	
Dn	000	register number	X			X	Dn
An	001	register number				X	An
An@	010	register number	X	X	X	X	(An)
An@+	011	register number	X	X		X	(An)+
An@-	100	register number	X	X		X	-(An)
An@(d)	101	register number	X	X	X	X	d(An)
An@(d, ix)	110	register number	X	X	X	X	d(An, Ri)
xxx.W	111	000	X	X	X	X	xxx
xxx.L	111	001	X	X	X	X	xxxxxx
PC@(d)	111	010	X	X	X		PC relative
PC@(d, ix)	111	011	X	X	X		PC rel. + Ri
#xxx	111	100	X	X			#xxx

The status register addressing mode is not permitted unless it is explicitly mentioned as a legal addressing mode.

These categories may be combined, so that additional, more restrictive, classifications may be defined. For example, the instruction descriptions use such classifications as alterable memory or data alterable. The former refers to those addressing modes which are both alterable and memory addresses, and the latter refers to addressing modes which are both data and alterable.

B.2 INSTRUCTION DESCRIPTION

The formats of each instruction are given in the following pages. Figure B-1 illustrates what information is given.

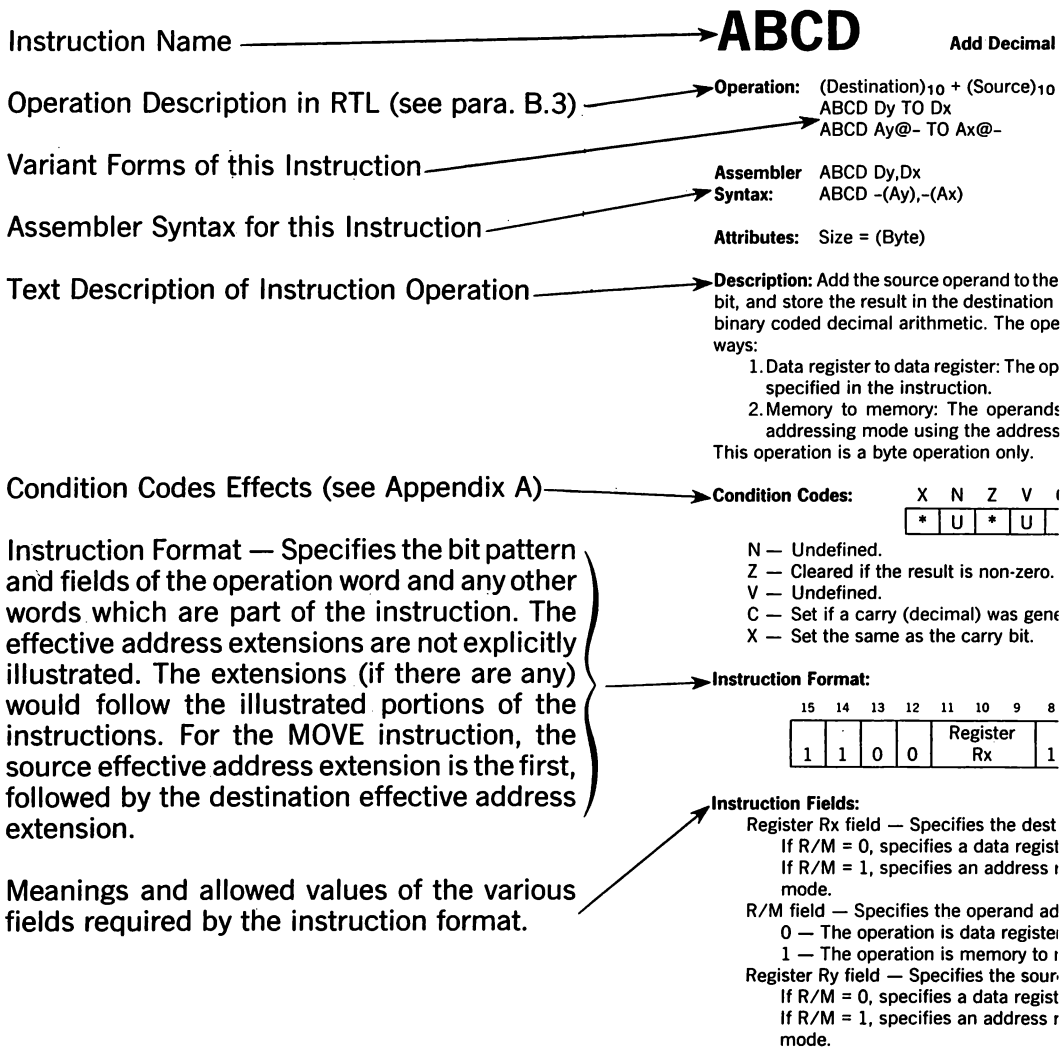

Instruction Name
ABCD
Add Decimal
Operation Description in RTL (see para. B.3)
Operation: (Destination)10 + (Source)10
ABCD Dy TO Dx
ABCD Ay@- TO Ax@-
Variant Forms of this Instruction
Assembler Syntax for this Instruction
Assembler Syntax: ABCD Dy,Dx
ABCD -(Ay),-(Ax)
Attributes: Size = (Byte)
Text Description of Instruction Operation
Description: Add the source operand to the
bit, and store the result in the destination
binary coded decimal arithmetic. The ope
ways:
1. Data register to data register: The op
specified in the instruction.
2. Memory to memory: The operands
addressing mode using the address
This operation is a byte operation only.
Condition Codes Effects (see Appendix A)
Condition Codes:
X N Z V
* U * U
N — Undefined.
Z — Cleared if the result is non-zero.
V — Undefined.
C — Set if a carry (decimal) was gene
X — Set the same as the carry bit.
Instruction Format — Specifies the bit pattern and fields of the operation word and any other words which are part of the instruction. The effective address extensions are not explicitly illustrated. The extensions (if there are any) would follow the illustrated portions of the instructions. For the MOVE instruction, the source effective address extension is the first, followed by the destination effective address extension.
Instruction Format:
15 14 13 12 11 10 9 8
1 1 0 0 Register Rx 1
Meanings and allowed values of the various fields required by the instruction format.
Instruction Fields:
Register Rx field — Specifies the dest
If R/M = 0, specifies a data regist
If R/M = 1, specifies an address r
mode.
R/M field — Specifies the operand ad
0 — The operation is data registe
1 — The operation is memory to r
Register Ry field — Specifies the sour
If R/M = 0, specifies a data regist
If R/M = 1, specifies an address r
mode.

Figure B-1. Instruction Description Format

B.3 REGISTER TRANSFER LANGUAGE DEFINITIONS

The following register transfer language definitions are used for the operation description in the details of the instruction set.

OPERANDS:

An — address register
Dn — data register
Rn — any data or address register
PC — program counter
SR — status register
CCR — condition codes (low order byte of status register)
SSP — supervisor stack pointer
USP — user stack pointer
SP — active stack pointer (equivalent to A7)
X — extend operand (from condition codes)
Z — zero condition code
V — overflow condition code

Immediate Data — immediate data from the instruction
d — address displacement
Source — source location
Destination — destination location
Vector — location of exception vector

SUBFIELDS AND QUALIFIERS:

<bit> OF <operand> selects a single bit of the operand
<operand> [<bit number> : <bit number>] selects a subfield of an operand
(<operand>) the contents of the referenced location
<operand>$_{10}$ the operand is binary coded decimal; operations are to be performed in decimal.
<operand> @ <mode> the register indirect operator which indicates that the operand register points to the memory location of the instruction operand. The optional mode qualifiers are –, +, (d) and (d, ix); these are explained in Chapter 2.

OPERATIONS: — operations are grouped into binary, unary, and other.
Binary — these operations are written <operand> <op> <operand> where <op> is one of the following:

→ the left operand is moved to the location specified by the right operand
↔ the contents of the two operands are exchanged
+ the operands are added
– the right operand is subtracted from the left operand
* the operands are multiplied
/ the first operand is divided by the second operand
∧ the operands are logically ANDed
v the operands are logically ORed
⊕ the operands are logically exclusively ORed
< relational test, true if left operand is less than right operand
> relational test, true if left operand is greater than right operand
≠ relational test, true if left operand is not equal to right operand
shifted by } the left operand is shifted or rotated by the number of positions specified by the right
rotated by } operand

Unary:

~<operand> the operand is logically complemented
<operand> sign-extended the operand is sign extended, all bits of the upper half are made equal to high order bit of the lower half.
<operand> tested the operand is compared to 0, the results are used to set the condition codes

Other:

TRAP equivalent to PC →SSP@–; SR →SSP@–; (vector) →PC
STOP enter the stopped state, waiting for interrupts
If <condition> then <operations> else <operations> The condition is tested. If true, the operations after the then are performed. If the condition is false and the optional else clause is present, the operations after the else are performed. If the condition is false and the optional else clause is absent, the instruction performs no operation.

ABCD Add Decimal with Extend ABCD

Operation: $(Destination)_{10} + (Source)_{10} + X \rightarrow Destination$
ABCD Dy TO Dx
ABCD Ay@- TO Ax@-

Assembler Syntax: ABCD Dy,Dx
ABCD -(Ay),-(Ax)

Attributes: Size = (Byte)

Description: Add the source operand to the destination operand along with the extend bit, and store the result in the destination location. The addition is performed using binary coded decimal arithmetic. The operands may be addressed in two different ways:

1. Data register to data register: The operands are contained in the data registers specified in the instruction.
2. Memory to memory: The operands are addressed with the predecrement addressing mode using the address registers specified in the instruction.

This operation is a byte operation only.

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

N — Undefined.
Z — Cleared if the result is non-zero. Unchanged otherwise.
V — Undefined.
C — Set if a carry (decimal) was generated. Cleared otherwise.
X — Set the same as the carry bit.

Instruction Format:

15	14	13	12	11 10 9	8	7	6	5	4	3	2 1 0
1	1	0	0	Register Rx	1	0	0	0	0	R/M	Register Ry

Instruction Fields:

Register Rx field — Specifies the destination register:
If R/M = 0, specifies a data register.
If R/M = 1, specifies an address register for the predecrement addressing mode.

R/M field — Specifies the operand addressing mode:
0 — The operation is data register to data register.
1 — The operation is memory to memory.

Register Ry field — Specifies the source register:
If R/M = 0, specifies a data register.
If R/M = 1, specifies an address register for the predecrement addressing mode.

Operation: (Destination) + (Source) → Destination
ADD <ea> TO Dn
ADD Dn TO <ea>

Assembler Syntax: ADD <ea>,Dn
ADD Dn,<ea>

Attributes: Size = (Byte, Word, Long)

Description: Add the source operand to the destination operand, and store the result in the destination location. The size of the operation may be specified to be byte, word or long. The mode of the instruction indicates which operand is the source and which is the destination as well as the operand size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Set if an overflow is generated. Cleared otherwise.
C — Set if a carry is generated. Cleared otherwise.
X — Set the same as the carry bit.

Instruction Format:

15	14	13	12	11 10 9	8 7 6	5 4 3 2 1 0
1	1	0	1	Register	Op-Mode	Effective Address

Instruction Fields:
Register field — Specifies any of the eight data registers.
Op-Mode field —

Byte	Word	Long	Operation
000	001	010	(<Dn>) + (<ea>) → <Dn>
100	101	110	(<ea>) + (<Dn>) → <ea>

Effective Address field — Determines addressing mode:
a. If the location specified is a source operand, then all addressing modes are allowed. If the operation size is byte, then the address register direct addressing mode is not allowed.
b. If the location specified is a destination operand, then only alterable memory addressing modes are allowed.

ADDA Add Address ADDA

Operation: (Destination) + (Source) → Destination
ADD <ea> TO An

Assembler Syntax: ADD <ea>,An

Attributes: Size = (Word, Long)

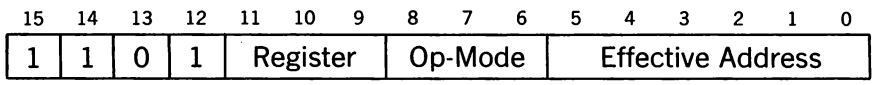
Description: Add the source operand to the destination address register, and store the result in the address register. The size of the operation may be specified to be word or long.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:



Instruction Fields:

Register field — Specifies any of the eight address registers. This is always the destination.

Op-Mode field — Specifies the size of the operation:

011 — word operation. The source operand is sign-extended to a long operand and the operation is performed on the address register using all 32 bits.

111 — long operation.

Effective Address field — Specifies the source operand. All addressing modes are allowed.

Operation: (Destination) + Immediate Data →Destination
ADD #<data> TO <ea>

Assembler Syntax: ADDI #<data>,<ea>

Attributes: Size = (Byte, Word, Long)

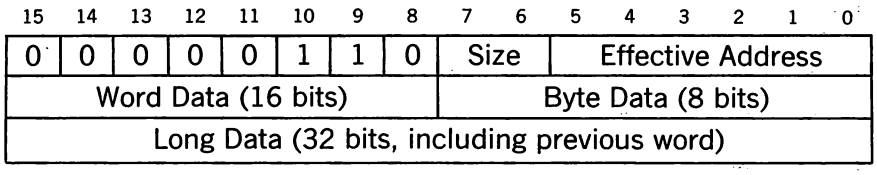
Description: Add the immediate data to the destination operand, and store the result in the destination location. The size of the operation may be specified to be byte, word or long. The size of the immediate data matches the operation size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Set if an overflow is generated. Cleared otherwise.
C — Set if a carry is generated. Cleared otherwise.
X — Set the same as the carry bit.

Instruction Format:


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 1 0 Size Effective Address
Word Data (16 bits)
Byte Data (8 bits)
Long Data (32 bits, including previous word)

Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — byte operation.
- 01 — word operation.
- 10 — long operation.

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed.

Immediate field — (Data immediately following the instruction):
- If size = 00, then the data is the low order byte of the immediate word.
- If size = 01, then the data is the entire immediate word.
- If size = 10, then the data is the next two immediate words.

ADDQ Add Quick ADDQ

Operation: (Destination) + Immediate Data →Destination
ADDQ #<data> TO <ea>

Assembler Syntax: ADDQ #<data>,<ea>

Attributes: Size = (Byte, Word, Long)

Description: Add the immediate data to the operand at the destination location. The data range is from 1 to 8. The size of the operation may be specified to be byte, word or long. Word and long operations are also allowed on the address registers.

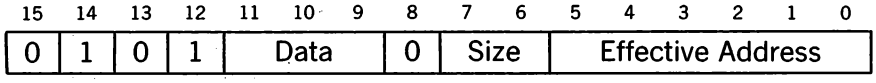
Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Set if an overflow is generated. Cleared otherwise.
C — Set if a carry is generated. Cleared otherwise.
X — Set the same as the carry bit.

The condition codes are not affected if an addition to an address register is made.

Instruction Format:



Instruction Fields:

Data field — Three bits of immediate data, 0, 1-7 representing a range of 8, 1 to 7 respectively.

Size field — Specifies the size of the operation:
- 00 — byte operation.
- 01 — word operation.
- 10 — long operation.

Effective Address field — Specifies the destination location. Only alterable addressing modes are allowed. If the operation size is byte, the address register direct addressing mode is not allowed.

Operation: (Destination) + (Source) + X → Destination
ADDX Dy TO Dx
ADDX Ay@- TO Ax@-

Assembler Syntax: ADDX Dy,Dx
ADDX -(Ay),-(Ax)

Attributes: Size = (Byte, Word, Long)

Description: Add the source operand to the destination operand along with the extend bit, and store the result in the destination location. The operands may be addressed in two different ways:

1. Data register to data register: the operands are contained in data registers specified in the instruction.
2. Memory to memory: the operands are addressed with the predecrement addressing mode using the address registers specified in the instruction.

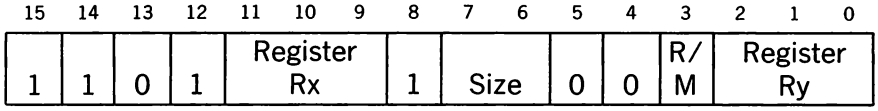
The size of the operation may be specified to be byte, word or long.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set if the result is negative. Cleared otherwise.
Z — Cleared if the result is non-zero. Unchanged otherwise.
V — Set if an overflow is generated. Cleared otherwise.
C — Set if a carry is generated. Cleared otherwise.
X — Set the same as the carry bit.

Instruction Format:



Instruction Fields:

Register Rx field — Specifies the destination register:
If R/M = 0, specifies a data register.
If R/M = 1, specifies an address register for the predecrement addressing mode.

Size field — Specifies the size of the operation:
00 — byte operation.
01 — word operation.
10 — long operation.

R/M field — Specifies the operand addressing mode:

0 — The operation is data register to data register.

1 — The operation is memory to memory.

Register Ry field — Specifies the source register:

If R/M = 0, specifies a data register.

If R/M = 1, specifies an address register for the predecrement addressing mode.

Operation: (Destination) ∧ (Source) →Destination
AND <ea> TO Dn
AND Dn TO <ea>

Assembler Syntax: AND <ea>,Dn
AND Dn,<ea>

Attributes: Size = (Byte, Word, Long)

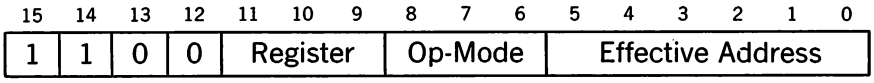
Description: AND the source operand to the destination operand, and store the result in the destination location. The size of the operation may be specified to be byte, word or long. The contents of an address register may not be used as an operand.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N — Set if the most significant bit of the result is set. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Always cleared.
C — Always cleared.
X — Not affected.

Instruction Format:



Instruction Fields:
Register field — Specifies any of the eight data registers.
Op-Mode field —

Byte	Word	Long	Operation
000	001	010	(<Dn>) ∧ (<ea>) →<Dn>
100	101	110	(<ea>) ∧ (<Dn>) →<ea>

Effective Address field — Determines addressing mode:
If the location specified is a source operand then only data addressing modes are allowed.
If the location specified is a destination operand then only alterable memory addressing modes are allowed.

Operation: (Destination) Λ Immediate Data →Destination
AND #<data> TO <ea>

Assembler Syntax: ANDI #<data>,<ea>

Attributes: Size = (Byte, Word, Long)

Description: AND the source operand to the destination operand, and store the result in the destination location. The size of the operation may be specified to be byte, word or long. The size of the immediate data matches the operation size. Access to the status register is available in two modes:

1. If the operation size is byte, the operation affects only the low order byte of the status register, which contains the condition codes.
2. If the operation size is word, the operation is performed on the whole status register. This is a privileged operation.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N — Set if the most significant bit of the result is set. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Always cleared.
C — Always cleared.
X — Not affected.
When accessing the status register the condition codes are affected according to the operation.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	Size		Effective Address					
Word Data (16 bits)								Byte Data (8 bits)							
Long Data (32 bits, including previous word)															

Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — byte operation.
- 01 — word operation.
- 10 — long operation.

Effective Address field — Specifies the destination operand. Only data alterable or the status register addressing modes are allowed.

Immediate field — (Data immediately following the instruction):
- If size = 00, then the data is the low order byte of the immediate word.
- If size = 01, then the data is the entire immediate word.
- If size = 10, then the data is the next two immediate words.

Operation: (Destination) Shifted by <count> →Destination
ASd Dy BY Dx
ASd Dy BY #<data>
ASd <ea>

Assembler Syntax:
ASd Dx,Dy
ASd #<data>,Dy
ASd <ea>

Attributes: Size = (Byte, Word, Long)

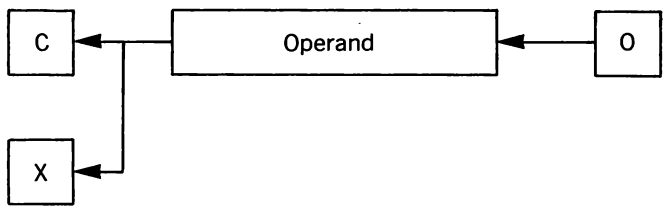
Description: Arithmetically shift the bits of the operand in the direction specified. The carry bit receives the last bit shifted out of the operand. The shift count for the shifting of a register may be specified in two different ways:

1. Immediate: the shift count is specified in the instruction (shift range, 1-8).
2. Register: the shift count is contained in a data register specified in the instruction.

The size of the operation may be specified to be byte, word or long. The content of memory may be shifted one bit only and the operand size is restricted to a word.

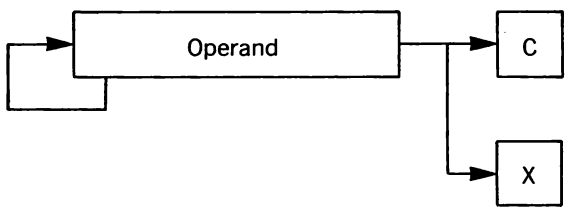
For ASL, the operand is shifted left; the number of positions shifted is the shift count. Bits shifted out of the high order bit go to both the carry and the extend bits; zeroes are shifted into the low order bit. The overflow bit indicates if any sign changes occur during the shift.

ASL:



For ASR, the operand is shifted right; the number of positions shifted is the shift count. Bits shifted out of the low order bit go to both the carry and the extend bits; the sign bit is replicated into the high order bit.

ASR:



Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set if the most significant bit of the result is set. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Set if the most significant bit is changed at any time during the shift operation. Cleared otherwise.
C — Set according to the last bit shifted out of the operand. Cleared for a shift count of zero.
X — Set according to the last bit shifted out of the operand. Unaffected for a shift count of zero.

Instruction Format (Register Shifts):

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	Count/ Register	dr	Size	i/r	0	0	Register

Instruction Fields (Register Shifts):

Count/Register field — Specifies shift count or register where count is located:
If i/r = 0, the shift count is specified in this field. The values 0, 1-7 represent a range of 8, 1 to 7 respectively.
If i/r = 1, the shift count (modulo 64) is contained in the data register specified in this field.

dr field — Specifies the direction of the shift:
0 — shift right.
1 — shift left.

Size field — Specifies the size of the operation:
00 — byte operation.
01 — word operation.
10 — long operation.

i/r field —
If i/r = 0, specifies immediate shift count.
If i/r = 1, specifies register shift count.

Register field — Specifies a data register whose content is to be shifted.

Instruction Format (Memory Shifts):

15	14	13	12	11	10	9	8	7	6	5 4 3 2 1 0
1	1	1	0	0	0	0	dr	1	1	Effective Address

Instruction Fields (Memory Shifts):

dr field — Specifies the direction of the shift:
0 — shift right.
1 — shift left.

Effective Address field — Specifies the operand to be shifted. Only memory alterable addressing modes are allowed.

Operation: If cc then PC + d →PC
Bcc <label>

Assembler Syntax: Bcc <label>

Attributes: Size = (Byte, Word)

Description: If the specified condition is met, program execution continues at location (PC) + displacement. Displacement is a two's complement integer which counts the relative distance in bytes. The value in PC is the current (instruction) location plus two. If the 8-bit displacement in the instruction word is zero, then the 16-bit displacement (word immediately following the instruction) is used. "cc" may specify the following conditions:

CC — carry clear
CS — carry set
EQ — equal
GE — greater or equal
GT — greater
HI — high
LE — less or equal
LS — low or same
LT — less
MI — minus
NE — not equal
PL — plus
VC — no overflow
VS — overflow

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	Condition				8-bit Displacement							
16-bit Displacement if 8-bit Displacement = 0															

Instruction Fields:

Condition field — One of fourteen conditions:

CC — 0100	LS — 0011
CS — 0101	LT — 1101
EQ — 0111	MI — 1011
GE — 1100	NE — 0110
GT — 1110	PL — 1010
HI — 0010	VC — 1000
LE — 1111	VS — 1001

8-bit Displacement field — Two's complement integer specifying the relative distance (in bytes) between the branch instruction and the next instruction to be executed if the condition is met.

16-bit Displacement field — Allows a larger displacement than 8 bits. Used only if the 8-bit displacement is equal to zero.

BCHG Test a Bit and Change BCHG

Operation: ~(<bit number> OF Destination) →Z;
~(<bit number> OF Destination) → <bit number > OF Destination
BCHG Dn OF <ea>
BCHG #<data> OF <ea>

Assembler Syntax: BCHG Dn,<ea>
BCHG #<data>,<ea>

Attributes: Size = (Word, Long)

Description: A bit in the destination operand is tested and the state of the specified bit is reflected in the Z condition code. After the test, the state of the specified bit is changed in the destination. If a data register is the destination, then the bit numbering is modulo 32 allowing bit manipulation on all bits in a data register. If a memory location is the destination, a byte is read from that location, the bit operation performed using the bit number modulo 8, and the byte written back to the location. The bit number for this operation may be specified in two different ways:

1. Immediate — the bit number is specified in a second word of the instruction.
2. Register — the bit number is contained in a data register specified in the instruction.

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

N — Not affected.
Z — Set if the bit tested is zero. Cleared otherwise.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format (Bit Number Dynamic):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Register			1	0	1	Effective Address					

Instruction Fields (Bit Number Dynamic):

Register field — Specifies the data register whose content is the bit number.
Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed.

Instruction Format (Bit Number Static):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	Effective Address					
0	0	0	0	0	0	0	0	bit number							

Instruction Fields (Bit Number Static):

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed.

bit number field — Specifies the bit numbers.

Operation: ~(<bit number> OF Destination) → Z;
0 → <bit number> → OF Destination
BCLR Dn OF <ea>
BCLR #<data> OF <ea>

Assembler Syntax: BCLR Dn,<ea>
BCLR #<data>,<ea>

Attributes: Size = (Word, Long)

Description: A bit in the destination operand is tested and the state of the specified bit is reflected in the Z condition code. After the test, the specified bit is cleared in the destination. If a data register is the destination, then the bit numbering is modulo 32 allowing bit manipulation on all bits in a data register. If a memory location is the destination, a byte is read from that location, the bit operation performed using the bit number modulo 8, and the byte written back to the location. The bit number for this operation may be specified in two different ways:

1. Immediate — the bit number is specified in a second word of the instruction.
2. Register — the bit number is contained in a data register specified in the instruction.

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

N — Not affected.
Z — Set if the bit tested is zero. Cleared otherwise.
V — Not affected.
C — Not affected.
X — Not affected.

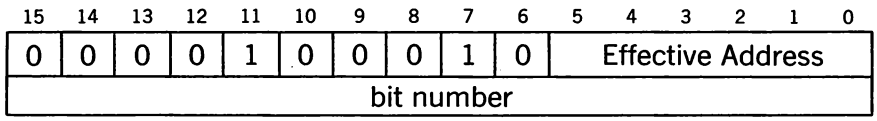
Instruction Format (Bit Number Dynamic):

15	14	13	12	11 10 9	8	7	6	5 4 3 2 1 0
0	0	0	0	Register	1	1	0	Effective Address

Instruction Fields (Bit Number Dynamic):

Register field — Specifies the data register whose content is the bit number.
Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed.

Instruction Format (Bit Number Static):



Instruction Fields (Bit Number Static):

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed.

bit number field — Specifies the bit number.

Operation: PC + d → PC
BRA <label>

Assembler Syntax: BRA <label>

Attributes: Size = (Byte, Word)

Description: Program execution continues at location (PC) + displacement. Displacement is a two's complement integer which counts the relative distance in bytes. The value in PC is the current (instruction) location plus two. If the 8-bit displacement in the instruction word is zero, then the 16-bit displacement (word immediately following the instruction) is used.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	8-bit Displacement							
16-bit Displacement if 8-bit Displacement = 0															

Instruction Fields:

8-bit Displacement field — Two's complement integer specifying the relative distance (in bytes) between the branch instruction and the next instruction to be executed if the condition is met.

16-bit Displacement field — Allows a larger displacement than 8 bits. Used only if the 8-bit displacement is equal to zero.

BSET — Test a Bit and Set — BSET

Operation: ~(<bit number>) OF Destination → Z
1 → <bit number> OF Destination
BSET Dn, OF <ea>
BSET #<data> OF <ea>

Assembler Syntax: BSET Dn, <ea>
BSET #<data>,<ea>

Attributes: Size = (Word, Long)

Description: A bit in the destination operand is tested and the state of the specified bit is reflected in the Z condition code. After the test, the specified bit is set in the destination. If a data register is the destination, then the bit numbering is modulo 32, allowing bit manipulation on all bits in a data register. If a memory location is the destination, a byte is read from that location, the bit operation performed using the bit number modulo 8, and the byte written back to the location. The bit number for this operation may be specified in two different ways:

1. Immediate — the bit number is specified in a second word of the instruction.
2. Register — the bit number is contained in a data register specified in the instruction.

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

N — Not affected.
Z — Set if the bit tested is zero. Cleared otherwise.
V — Not affected.
C — Not affected.
X — Not affected.

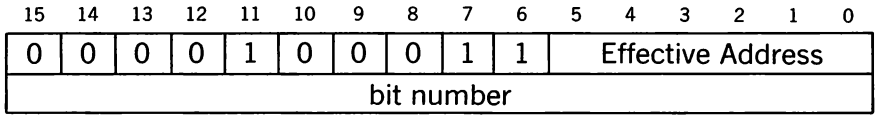
Instruction Format (Bit Number Dynamic):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Register			1	1	1	Effective Address					

Instruction Fields (Bit Number Dynamic):

Register field — Specifies the data register whose content is the bit number.
Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed.

Instruction Format (Bit Number Static):



Instruction Fields (Bit Number Static):

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed.

bit number field — Specifies the bit number.

Operation: PC →SP@–; PC + d →PC
BSR <label>

Assembler Syntax: BSR <label>

Attributes: Size = (Byte, Word)

Description: The address of the instruction immediately following the BSR instruction is pushed onto the system stack. Program execution then continues at location (PC) + displacement. Displacement is a two's complement integer which counts the relative distance in bytes. The value in PC is the current (instruction) location plus two. If the 8-bit displacement in the instruction word is zero, then the 16-bit displacement (word immediately following the instruction) is used.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	1	8-bit Displacement							
16-bit Displacement if 8-bit Displacement = 0															

Instruction Fields:

8-bit Displacement field — Two's complement integer specifying the relative distance (in bytes) between the branch instruction and the next instruction to be executed if the condition is met.

16-bit Displacement field — Allows a larger displacement than 8 bits. Used only if the 8-bit displacement is equal to zero.

Operation: ~(<bit number> OF Destination) → Z
BTST Dn OF <ea>
BTST #<data> OF <ea>

Assembler Syntax: BTST Dn,<ea>
BTST #<data>,<ea>

Attributes: Size = (Word, Long)

Description: A bit in the destination operand is tested and the state of the specified bit is reflected in the Z condition code. If a data register is the destination, then the bit numbering is modulo 32, allowing bit manipulation on all bits in a data register. If a memory location is the destination, a byte is read from that location, and the bit operation performed using the bit number modulo 8. The bit number for this operation may be specified in two different ways:

1. Immediate — the bit number is specified in a second word of the instruction.
2. Register — the bit number is contained in a data register specified in the instruction.

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

N — Not affected.
Z — Set if the bit tested is zero. Cleared otherwise.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format (Bit Number Dynamic):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Register			1	0	0	Effective Address					

Instruction Fields (Bit Number Dynamic):
Register field — Specifies the data register whose content is the bit number.
Effective Address field — Specifies the destination location. Only data addressing modes are allowed.

Instruction Format (Bit Number Static):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	Effective Address					
bit number															

Instruction Fields (Bit Number Static):
Effective Address field — Specifies the destination location. Only data addressing modes are allowed, except the immediate addressing mode is not allowed.
bit number field — Specifies the bit number.

Operation: If Dn <0 or Dn> (<ea>) then TRAP
CHK Dn WITH <ea>

Assembler Syntax: CHK <ea>,Dn

Attributes: Size = (Word)

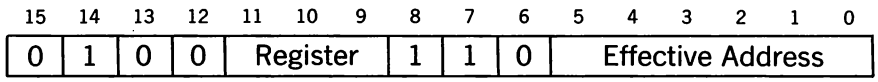
Description: The content of the low order word in the data register specified in the instruction is examined, and compared to the upper bound. The upper bound is a two's complement integer. If the register value is less than zero or greater than the upper bound contained in the operand word, then the processor initiates exception processing. The vector number is generated to reference the CHK instruction exception vector.

Condition Codes:

X	N	Z	V	C
—	*	U	U	U

N — Set if Dn <0; cleared if Dn> (<ea>). Undefined otherwise.
Z — Undefined.
V — Undefined.
C — Undefined.
X — Not affected.

Instruction Format:



Instruction Fields:
Register field — Specifies the data register whose content is checked.
Effective Address field — Specifies the upper bound operand word. Only data addressing modes are allowed.

Operation: 0 → Destination
CLR <ea>

Assembler Syntax: CLR <ea>

Attributes: Size = (Byte, Word, Long)

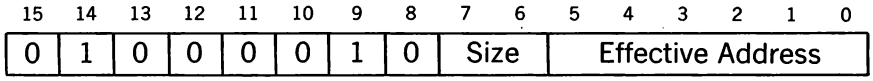
Description: The destination is cleared to all zero bits. The size of the operation may be specified to be byte, word or long.

Condition Codes:

X	N	Z	V	C
—	0	1	0	0

N — Always cleared.
Z — Always set.
V — Always cleared.
C — Always cleared.
X — Not affected.

Instruction Format:



Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — byte operation.
- 01 — word operation.
- 10 — long operation.

Effective Address field — Specifies the destination location. Only data alterable addresses are allowed.

Operation: (Destination) - (Source)
CMP <ea> TO Dn

Assembler Syntax: CMP <ea>,Dn

Attributes: Size = (Byte, Word, Long)

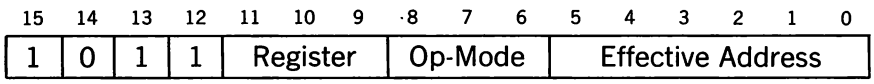
Description: Subtract the source operand from the destination operand, and set the condition codes according to the result; the destination location is not changed. The size of the operation may be specified to be byte, word or long.

Condition Codes:

X	N	Z	V	C
—	*	*	*	*

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Set if an overflow is generated. Cleared otherwise.
C — Set if a borrow is generated. Cleared otherwise.
X — Not affected.

Instruction Format:



Instruction Fields:

Register field — Specifies the destination data register.

Op-Mode field —

Byte	Word	Long	Operation
000	001	010	(<Dn>) - (<ea>)

Effective Address field — Specifies the source operand. All addressing modes are allowed. If the operation size is byte, then the address register direct addressing mode is not allowed.

CMPA Compare Address CMPA

Operation: (Destination) - (Source)
CMP <ea> TO An

Assembler Syntax: CMPA <ea>,An

Attributes: Size = (Word, Long)

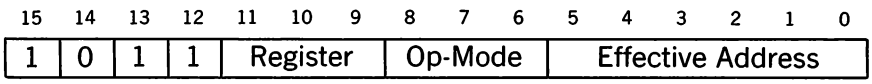
Description: Subtract the source operand from the destination address register, and set the condition codes according to the result; the address register is not changed. The size of the operation may be specified to be word or long.

Condition Codes:

X	N	Z	V	C
—	*	*	*	*

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Set if an overflow is generated. Cleared otherwise.
C — Set if a borrow is generated. Cleared otherwise.
X — Not affected.

Instruction Format:



Instruction Fields:

Register field — Specifies the destination address register.

Op-Mode field — Specifies the size of the operation:

011 — word operation. The source operand is sign-extended to a long operand and the operation is performed on the address register using all 32 bits.

111 — long operation.

Effective Address field — Specifies the source operand. All addressing modes are allowed.

Operation: (Destination) - Immediate Data
CMP #<data> TO <ea>

Assembler Syntax: CMPI #<data>,<ea>

Attributes: Size = (Byte, Word, Long)

Description: Subtract the immediate data from the destination operand, and set the condition codes according to the result; the destination location is not changed. The size of the operation may be specified to be byte, word or long. The size of the immediate data matches the operation size.

Condition Codes:

X	N	Z	V	C
—	*	*	*	*

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Set if an overflow is generated. Cleared otherwise.
C — Set if a borrow is generated. Cleared otherwise.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	Size		Effective Address					
Word Data (16 bits)								Byte Data (8 bits)							
Long Data (32 bits, including previous word)															

Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — byte operation.
- 01 — word operation.
- 10 — long operation.

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed.

Immediate field — (Data immediately following the instruction):
- If size = 00, then the data is the low order byte of the immediate word.
- If size = 01, then the data is the entire immediate word.
- If size = 10, then the data is the next two immediate words.

Operation: (Destination) - (Source)
CMP Ay@+ TO Ax@+

Assembler Syntax: CMPM (Ay)+,(Ax)+

Attributes: Size = (Byte, Word, Long)

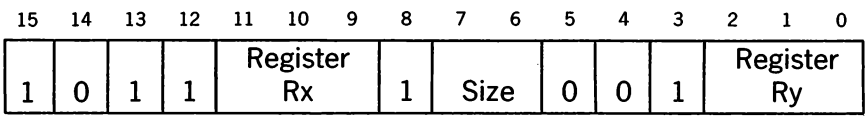
Description: Subtract the source operand from the destination operand, and set the condition codes according to the results; the destination location is not changed. The operands are always addressed with the postincrement addressing mode using the address registers specified in the instruction. The size of the operation may be specified to be byte, word or long.

Condition Codes:

X	N	Z	V	C
—	*	*	*	*

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Set if an overflow is generated. Cleared otherwise.
C — Set if a borrow is generated. Cleared otherwise.
X — Not affected.

Instruction Format:



Instruction Fields:

Register Rx field — (always the destination) Specifies an address register for the postincrement addressing mode.

Size field — Specifies the size of the operation:
- 00 — byte operation.
- 01 — word operation.
- 10 — long operation.

Register Ry field — (always the source) Specifies an address register for the postincrement addressing mode.

DB_{CC} Test Condition, Decrement and Branch DB_{CC}

Operation: If $\sim_{CC}$ then Dn - 1 → Dn; if Dn ≠ -1 then PC + d → PC
DB_{cc} Dn, <label>

Assembler Syntax: DB_{CC} Dn, <label>

Description: This instruction is a looping primitive of three parameters: a condition, a data register, and a displacement. The instruction first tests the condition to determine if the termination condition for the loop has been met, and if so, no operation is performed. If the termination condition is not true, the counter data register is decremented by one. If the result is -1, the counter is exhausted, and execution continues with the next instruction. If the result is not equal to -1, execution continues at the location indicated by the current value of PC plus the sign-extended 16-bit displacement. The value in PC is the address of the displacement word. "cc" may specify the following conditions:

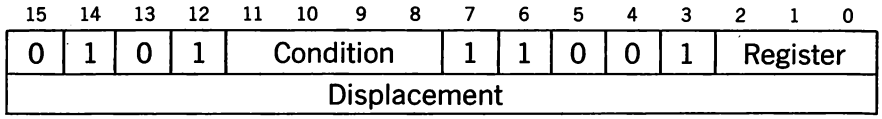
CC	0100 — carry clear	LS	0011 — low or same
CS	0101 — carry set	LT	1101 — less
EQ	0111 — equal	MI	1011 — minus
F*	0001 — never true	NE	0110 — not equal
GE	1100 — greater or equal	PL	1010 — plus
GT	1110 — greater	T	0000 — always true
HI	0010 — high	VC	1000 — no overflow
LE	1111 — less or equal	VS	1001 — overflow

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:



Instruction Fields:

Condition field — One of the sixteen conditions discussed in description.
Register field — Specifies the data register which is the counter.
Displacement field — Specifies the distance of the branch (in bytes).

* assembler accepts DBRA for this condition

Operation: (Destination)/(Source) →Destination
DIVS Dn BY <ea>

Assembler Syntax: DIVS <ea>,Dn

Attributes: Size = (Word)

Description: Divide the destination operand by the source operand and store the result in the destination. The destination operand is a long operand (32-bits) and the source operand is a word operand (16-bits). The operation is performed using signed arithmetic. The result is a 32-bit result such that:

1. The quotient is in the lower word (least signifcant 16-bits).
2. The remainder is in the upper word (most significant 16-bits).

The sign of the remainder is always the same as the dividend unless the remainder is equal to zero. Two special conditions may arise:

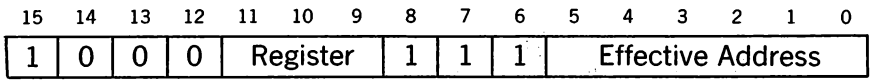
1. Division by zero causes a trap.
2. Overflow may be detected and set before completion of the instruction. If overflow is detected, the condition is flagged but the operands are unaffected.

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

N — Set if the quotient is negative. Cleared otherwise. Undefined if overflow.
Z — Set if the quotient is zero. Cleared otherwise. Undefined if overflow.
V — Set if division overflow is detected. Cleared otherwise.
C — Always cleared.
X — Not affected.

Instruction Format:



Instruction Fields:

Register field — Specifies any of the eight data registers. This field always specifies the destination operand.

Effective Address field — Specifies the source operand. Only data addressing modes are allowed.

Operation: (Destination)/(Source) →Destination
DIVU Dn BY <ea>

Assembler Syntax: DIVU <ea>,Dn

Attributes: Size = (Word)

Description: Divide the destination operand by the source operand and store the result in the destination. The destination operand is a long operand (32-bits) and the source operand is a word (16-bit) operand. The operation is performed using unsigned arithmetic. The result is a 32-bit result such that:

1. The quotient is in the lower word (least significant 16-bits).
2. The remainder is in the upper word (most significant 16-bits)

Two special conditions may arise:

1. Division by zero causes a trap.
2. Overflow may be detected and set before completion of the instruction. If overflow is detected, the condition is flagged but the operands are unaffected.

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

N — Set if the most significant bit of the quotient is set. Cleared otherwise. Undefined if overflow.
Z — Set if the quotient is zero. Cleared otherwise. Undefined if overflow.
V — Set if division overflow is detected. Cleared otherwise.
C — Always cleared.
X — Not affected.

Instruction Format:

15	14	13	12	11 10 9	8	7	6	5 4 3 2 1 0
1	0	0	0	Register	0	1	1	Effective Address

Instruction Fields:

Register field — Specifies any of the eight data registers. This field always specifies the destination operand.
Effective Address field — Specifies the source operand. Only data addressing modes are allowed.

Operation: (Destination) ⊕ (Source) → Destination
EOR Dn TO <ea>

Assembler Syntax: EOR Dn,<ea>

Attributes: Size = (Byte, Word, Long)

Description: Exclusive OR the source operand to the destination operand, and store the result in the destination location. The size of the operation may be specified to be byte, word or long. This operation is restricted to data registers as the source operand. The destination operand is specified in the effective address field.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N — Set if the most significant bit of the result is set. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Always cleared.
C — Always cleared.
X — Not affected.

Instruction Format:

15	14	13	12	11 10 9	8 7 6	5 4 3 2 1 0
1	0	1	1	Register	Op-Mode	Effective Address

Instruction Fields:
Register field — Specifies any of the eight data registers.
Op-Mode field —

Byte	Word	Long	Operation
100	101	110	(<ea>) ⊕ (<Dx>) → <ea>

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed.

Operation: (Destination) ⊕ Immediate Data → Destination
EORI #<data> TO <ea>

Assembler Syntax: EORI #<data>,<ea>

Attributes: Size = (Byte, Word, Long)

Description: Exclusive OR the immediate data to the destination operand, and store the result in the destination location. The size of the operation may be specified to be byte, word or long. The immediate data matches the operation size. Access to the status register is available in two modes:

1. If the operation size is byte, the operation affects only the low order byte of the status register which contains the condition codes.
2. If the operation size is word, the operation is performed on the whole status register. This is a privileged operation.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N — Set if the most significant bit of the result is set. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Always cleared.
C — Always cleared.
X — Not affected.
When accessing the status register, the condition codes are affected according to the operation.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	Size		Effective Address					
Word Data (16 bits)								Byte Data (8 bits)							
Long Data (32 bits, including previous word)															

Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — byte operation.
- 01 — word operation.
- 10 — long operation.

Effective Address field — Specifies the destination operand. Only data alterable or the status register addressing modes are allowed.

Immediate field — (Data immediately following the instruction):
- If size = 00, then the data is the low order byte of the immediate word.
- If size = 01, then the data is the entire immediate word.
- If size = 10, then the data is the next two immediate words.

Operation: Rx↔Ry
EXG Rx WITH Ry

Assembler Syntax: EXG Rx,Ry

Attributes: Size = (Long)

Description: Exchange the contents of two registers. This exchange is always a long (32-bit) operation. Exchange works in three modes:

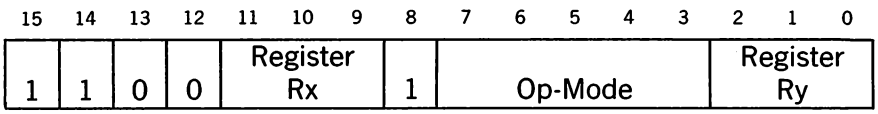
1. exchange data registers.
2. exchange address registers.
3. exchange a data register and an address register

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:



Instruction Fields:

Register Rx field — Specifies either a data register or an address register depending on the mode. If the exchange is between data and address registers, this field always specifies the data register.

Op-Mode field — Specifies whether exchanging:
01000 — data registers.
01001 — address registers.
10001 — data register and address register.

Register Ry field — Specifies either a data register or an address register depending on the mode. If the exchange is between data and address registers, this field always specifies the address register.

Operation: (Destination) Sign-extended →Destination
EXT Dn

Assembler Syntax: EXT Dn

Attributes: Size = (Word, Long)

Description: Extend the sign bit of a data register from a byte to a word or from a word to a long operand depending on the size selected. If the operation is word sized, bit [7] of the designated data register is copied to bits [15:8] of that data register. If the operation is long sized, bit [15] of the designated data register is copied to bits [31:16] of that data register.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Always cleared.
C — Always cleared.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	Op-Mode			0	0	0	Register		

Instruction Fields:
Op-Mode Field — Specifies the size of the sign-extension operation:
010 — Sign-extend low order byte of data register to word.
011 — Sign-extend low order word of data register to long.
Register field — Specifies the data register whose content is to be sign-extended.

Operation: Destination →PC
JMP <ea>

Assembler Syntax: JMP <ea>

Attributes: Unsized

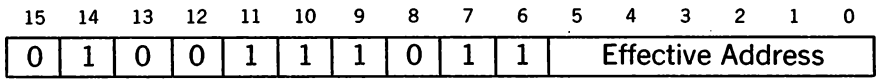
Description: Program execution continues at the address specified by the instruction. The address is specified by the control addressing modes.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:



Instruction Fields:

Effective Address field — Specifies the address of the next instruction. Only control addressing modes are allowed.

Operation: PC →SP@-; Destination →PC
JSR <ea>

Assembler Syntax: JSR <ea>

Attributes: Unsized

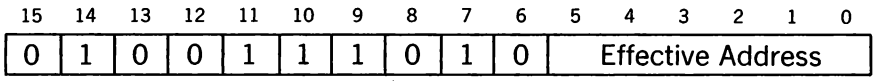
Description: The address of the instruction immediately following the JSR instruction is pushed onto the system stack. Program execution then continues at the address specified in the instruction.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:



Instruction Fields:

Effective Address field — Specifies the address of the next instruction. Only control addressing modes are allowed.

Operation: Destination → An
LEA <ea> INTO An

Assembler Syntax: LEA <ea>,An

Attributes: Size = (Long)

Description: The effective address is loaded into the specified address register. All 32 bits of the address register are affected by this instruction.

Condition Codes:

X	Ń	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	Register			1	1	1	Effective Address					

Instruction Fields:

Register field — Specifies the address register which is to be loaded with the effective address.

Effective Address field — Specifies the address to be loaded into the address register. Only control addressing modes are allowed.

Operation: An →SP@-; SP →An; SP + d →SP
LINK An, #<displacement>

Assembler Syntax: LINK An,#<displacement>

Attributes: Unsized

Description: The current content of the specified address register is pushed onto the stack. After the push, the address register is loaded from the updated stack pointer. Finally, the sign-extended displacement is added to the stack pointer. The content of the address register occupies two words on the stack.
Note: LINK and UNLK can be used to maintain a linked list of local data and parameter areas on the stack for nested subroutine calls.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	0	Register		
Displacement															

Instruction Fields:
Register field — Specifies the address register through which the link is to be constructed.
Displacement field — Specifies the two's complement integer which is to be added to the stack pointer.

Operation: (Destination) Shifted by <count> → Destination
LSd Dy BY Dx
LSd Dy BY #<data>
LSd <ea>

Assembler Syntax:
LSd Dx,Dy
LSd #<data>,Dy
LSd <ea>

Attributes: Size = (Byte, Word, Long)

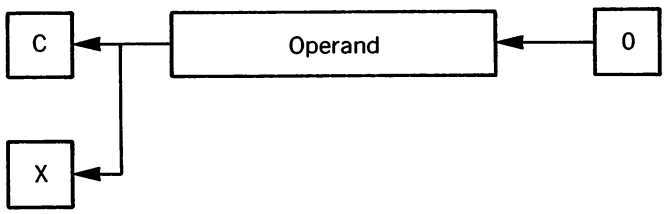
Description: Shift the bits of the operand in the direction specified. The carry bit receives the last bit shifted out of the operand. The shift count for the shifting of a register may be specified in two different ways:

1. Immediate — the shift count is specified in the instruction (shift range, 1-8).
2. Register — the shift count is contained in a data register specified in the instruction.

The size of the operation may be specified to be byte, word or long. The content of memory may be shifted one bit only where the operand size is restricted to a word.

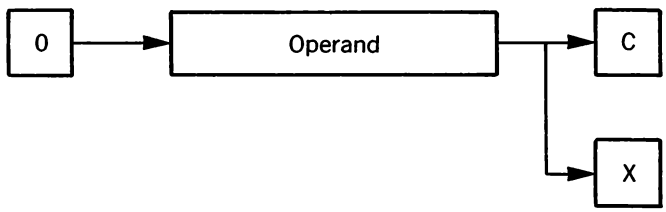
For LSL, the operand is shifted left; the number of positions shifted is the shift count. Bits shifted out of the high order bit go to both the carry and the extend bits; zeroes are shifted into the low order bit.

LSL:



For LSR, the operand is shifted right; the number of positions shifted is the shift count. Bits shifted out of the low order bit go to both the carry and the extend bits; zeroes are shifted into the high order bit.

LSR:



Condition Codes:

X	N	Z	V	C
*	*	*	0	*

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Always cleared.
C — Set according to the last bit shifted out of the operand. Cleared for a shift count of zero.
X — Set according to the last bit shifted out of the operand. Unaffected for a shift count of zero.

Instruction Format (Register Shifts):

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	Count/ Register	dr	Size	i/r	0	1	Register

Instruction Fields (Register Shifts):

Count/Register field —
If i/r = 0, the shift count is specified in this field. The values 0, 1-7 represent a range of 8, 1 to 7 respectively.
If i/r = 1, the shift count (modulo 64) is contained in the data register specified in this field.

dr field — Specifies the direction of the shift:
0 — shift right.
1 — shift left.

Size field — Specifies the size of the operation:
00 — byte operation.
01 — word operation.
10 — long operation.

i/r field —
If i/r = 0, specifies immediate shift count.
If i/r = 1, specifies register shift count.

Register field — Specifies a data register whose content is to be shifted.

Instruction Format (Memory Shifts):

15	14	13	12	11	10	9	8	7	6	5 4 3 2 1 0
1	1	1	0	0	0	1	dr	1	1	Effective Address

Instruction Fields (Memory Shifts):

dr field — Specifies the direction of the shift:
0 — shift right.
1 — shift left.

Effective Address field — Specifies the operand to be shifted. Only memory alterable addressing modes are allowed.

Operation: (Source) → Destination
MOVE <ea> TO <ea>

Assembler Syntax: MOVE <ea>,<ea>

Attributes: Size = (Byte, Word, Long)

Description: Move the content of the source to the destination location. The data is examined as it is moved, and the condition codes set accordingly. The size of the operation may be specified to be byte, word or long.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Always cleared.
C — Always cleared.
X — Not affected.

Instruction Format:

15	14	13 12	11 10 9	8 7 6	5 4 3	2 1 0
			Destination		Source	
0	0	Size	Register	Mode	Mode	Register

Instruction Fields:

Size field — Specifies the size of the operand to be moved:
- 01 — byte operation.
- 11 — word operation.
- 10 — long operation.

Destination Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed.

Source Effective Address field — Specifies the source operand. All addressing modes are allowed. If the operation size is byte, the address register direct addressing mode is not allowed.

MOVE to CCR

Move to Condition Codes

MOVE to CCR

Operation: (Source) →CCR
MOVE <ea> TO CCR

Assembler Syntax: MOVE <ea>,CCR

Attributes: Size = (Word)

Description: The content of the source operand is moved to the condition codes. The source operand is a word, but only the low order byte is used to update the condition codes.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set according to the source operand.
Z — Set according to the source operand.
V — Set according to the source operand.
C — Set according to the source operand.
X — Set according to the source operand.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	Effective Address					

Instruction Fields:

Effective Address field — Specifies the location of the source operand. Only data addressing modes are allowed.

MOVE to SR

Move to the Status Register

MOVE to SR

Operation: (Source) →SR
MOVE <ea> TO SR

Assembler Syntax: MOVE <ea>,SR

Attributes: Size = (Word)

Description: The content of the source operand is moved to the status register. The source operand is a word, and all bits of the status register are affected.
Note: This is a privileged instruction.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set according to the source operand.
Z — Set according to the source operand.
V — Set according to the source operand.
C — Set according to the source operand.
X — Set according to the source operand.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	1	1	Effective Address					

Instruction Fields:
Effective Address field — Specifies the location of the source operand. Only data addressing modes are allowed.

MOVE from SR

Move from the Status Register

MOVE from SR

Operation: SR → Destination
MOVE SR TO <ea>

Assembler Syntax: MOVE SR,<ea>

Attributes: Size = (Word)

Description: The content of the status register is moved to the destination location. The operand size is a word.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	Effective Address					

Instruction Fields:

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed.

MOVE USP

Move User Stack Pointer

MOVE USP

Operation: USP →An;
An →USP
MOVE USP TO An
MOVE An TO USP

Assembler Syntax: MOVE USP,An
MOVE An,USP

Attributes: Size = (Long)

Description: The contents of the user stack pointer are transferred to or from the specified address register. This instruction is useful for supervisor state programs.
Note: This is a privileged instruction.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	0	dr	Register		

Instruction Fields:
dr field — Specifies the direction of transfer:
0 — transfer the address register to the USP.
1 — transfer the USP to the address register.
Register field — Specifies the address register to or from which the user stack Pointer is to be transferred.

Operation: (Source) →Destination
MOVE <ea> TO An

Assembler Syntax: MOVEA <ea>,An

Attributes: Size = (Word, Long)

Description: Move the content of the source to the destination address register. The size of the operation may be specified to be word or long.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				Destination						Source					
0	0	Size		Register		0	0	1		Mode		Register			

Instruction Fields:

Size field — Specifies the size of the operand to be moved:
11 — Word operation. The source operand is sign-extended to a long operand and all 32 bits are loaded into the address register.
10 — Long operation.

Destination Register field — Specifies the destination address register.

Source Effective Address field — Specifies the location of the source operand. All addressing modes are allowed.

Operation: Registers → Destination
(Source) → Registers
MOVEM <register list> TO <ea>
MOVEM <ea> TO <register list>

Assembler Syntax: MOVEM <register list>,<ea>
MOVEM <ea>,<register list>

Attributes: Size = (Word, Long)

Description: Selected registers are transferred to or from consecutive memory locations starting at the location specified by the effective address. A register is transferred if the bit corresponding to that register is set in the mask field. The instruction selects how much of each register is transferred; either the entire long word can be moved, or just the low order word. In the case of a word transfer to the registers, each word is sign-extended to 32 bits and the resulting long word loaded into the associated register.

MOVEM allows three forms of address modes: the control modes, the predecrement mode, or the postincrement mode. If the effective address is in one of the control modes, the registers are transferred starting at the specified address and up through higher addresses. The order of transfer is from data register 0 to data register 7, then from address register 0 to address register 7.

If the effective address is in the predecrement mode, only a register to memory operation is allowed. The registers are stored starting at the specified address minus two and down through lower addresses. The order of storing is from address register 7 to address register 0, then from data register 7 to data register 0. The decremented address register is updated to contain the address of the last word stored.

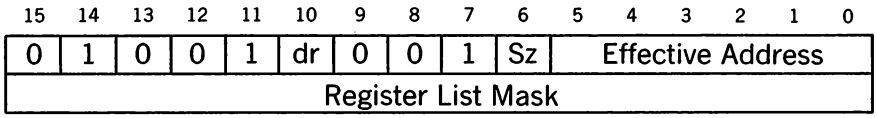
If the effective address is in the postincrement mode, only a memory to register operation is allowed. The registers are loaded starting at the specified address and up through higher addresses. The order of loading is the same as for the control mode addressing. The incremented address register is updated to contain the address of the last word loaded plus two.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:



Instruction Fields:

dr field:

Specifies the direction of the transfer:

0 — register to memory.

1 — memory to register.

Sz field — Specifies the size of the registers being transferred:

0 — word transfer.

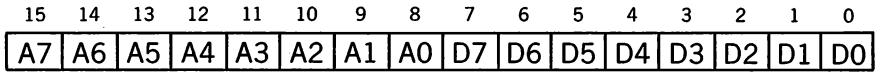
1 — long transfer.

Effective Address field — Specifies the memory address to or from which the registers are to be moved.

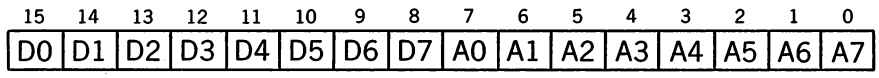
For register to memory transfer, only control alterable addressing modes, or the predecrement addressing mode are allowed.

For memory to register transfer, only control addressing modes, or the postincrement addressing mode are allowed.

Register List Mask field — Specifies which registers are to be transferred. The low order bit corresponds to the first register to be transferred; the high bit corresponds to the last register to be transferred. Thus, both for control modes and for the postincrement mode addresses, the mask correspondence is



while for the predecrement mode addresses, the mask correspondence is



MOVEP Move Peripheral Data MOVEP

Operation: (Source) → Destination
MOVEP Dx TO Ay@(d)
MOVEP Ay@(d) TO Dx

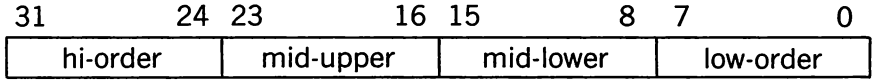
Assembler Syntax: MOVE Dx,d(Ay)
MOVEP d(Ay),Dx

Attributes: Size = (Word, Long)

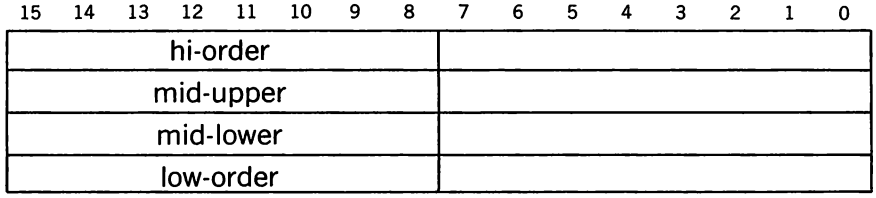
Description: Data is transferred between a data register and alternate bytes of memory, starting at the location specified, and incrementing by two. The high order byte of the data register is transferred first, and the low order byte is transferred last. The memory address is specified using the address register indirect plus displacement addressing mode. If the address is even, all the transfers are made on the high order half of the data bus; if the address is odd, all the transfers are made on the low order half of the data bus.

Example: Long transfer to/from an even address.

Byte organization in register

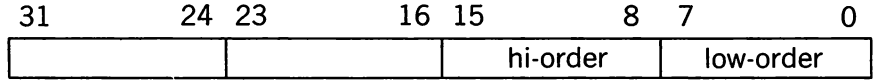


Byte organization in memory (low address at top)

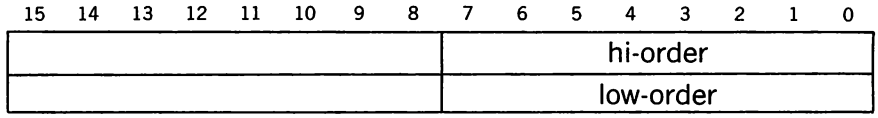


Example: Word transfer to/from an odd address.

Byte organization in register



Byte organization in memory (low address at top)

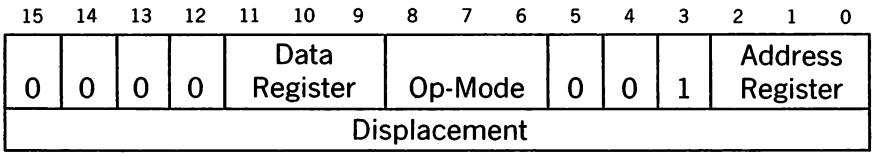


Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:



Instruction Fields:

Data Register field — Specifies the data register to or from which the data is to be transferred.

Op-Mode field — Specifies the direction and size of the operation:

100 — transfer word from memory to register.
101 — transfer long from memory to register.
110 — transfer word from register to memory.
111 — transfer long from register to memory.

Address Register field — Specifies the address register which is used in the address register indirect plus displacement addressing mode.

Displacement field — Specifies the displacement which is used in calculating the operand address.

MOVEQ Move Quick MOVEQ

Operation: Immediate Data → Destination
MOVEQ #<data> TO Dn

Assembler Syntax: MOVEQ #<data>,Dn

Attributes: Size = (Long)

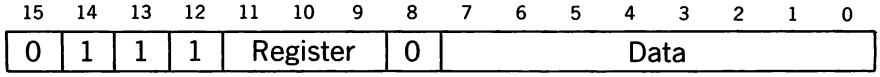
Description: Move immediate data to a data register. The data is contained in an 8-bit field within the operation word. The data is sign-extended to a long operand, and all 32 bits are transferred to the data register.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Always cleared.
C — Always cleared.
X — Not affected.

Instruction Format:



Instruction Fields:
Register field — Specifies the data register to be loaded.
Data field — 8 bits of data which are sign extended to a long operand.

Operation: (Destination) * (Source) →Destination
MULS Dn By <ea>

Assembler Syntax: MULS <ea>,Dn

Attributes: Size = (Word)

Description: Multiply two signed 16-bit operands yielding a 32-bit signed result. The operation is performed using signed arithmetic. A register operand is taken from the low order word, the upper word is unused. All 32 bits of the product are saved in the destination data register.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Always cleared.
C — Always cleared.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Register			1	1	1	Effective Address					

Instruction Fields:

Register field — Specifies one of the data registers. This field always specifies the destination.

Effective Address field — Specifies the source operand. Only data addressing modes are allowed.

Operation: (Destination) * (Source) →Destination
MULU Dn BY <ea>

Assembler Syntax: MULU <ea>,Dn

Attributes: Size = (Word)

Description: Multiply two unsigned 16-bit operands yielding a 32-bit unsigned result. The operation is performed using unsigned arithmetic. A register operand is taken from low order word; the upper word is unused. All 32 bits of the product are saved in the destination data register.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N — Set if the most significant bit of the result is set. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Always cleared.
C — Always cleared.
X — Not affected.

Instruction Format:

15	14	13	12	11 10 9	8	7	6	5 4 3 2 1 0
1	1	0	0	Register	0	1	1	Effective Address

Instruction Fields:
Register field — Specifies one of the data registers. This field always specifies the destination.
Effective Address field — Specifies the source operand. Only data addressing modes are allowed.

Operation: 0 - (Destination)$_{10}$ - X →Destination
NBCD <ea>

Assembler Syntax: NBCD <ea>

Attributes: Size = (Byte)

Description: The operand addressed as the destination and the extend bit are subtracted from zero. The operation is performed using decimal arithmetic. The result is saved in the destination location. This instruction produces the ten's complement of the destination if the extend bit is clear, the nine's complement if the extend bit is set. This is a byte operation only.

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

N — Undefined.
Z — Cleared if the result is non-zero. Unchanged otherwise.
V — Undefined.
C — Set if a borrow (decimal) was generated. Cleared otherwise.
X — Set the same as the carry bit.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	Effective Address					

Instruction Fields:

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed.

Operation: 0 - (Destination) →Destination
NEG <ea>

Assembler Syntax: NEG <ea>

Attributes: Size = (Byte, Word, Long)

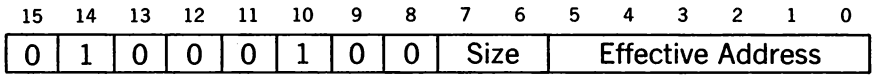
Description: The operand addressed as the destination is subtracted from zero. The result is stored in the destination location. The size of the operation may be specified to be byte, word, or long.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Set if an overflow is generated. Cleared otherwise.
C — Set if a borrow is generated. Cleared otherwise.
X — Set the same as the carry bit.

Instruction Format:



Instruction Fields:
Size field — Specifies the size of the operation:
00 — byte operation.
01 — word operation.
10 — long operation.
Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed.

Operation: 0 - (Destination) - X →Destination
NEGX <ea>

Assembler Syntax: NEGX <ea>

Attributes: Size = (Byte, Word, Long)

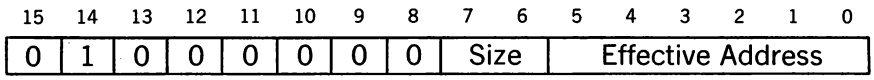
Description: The operand addressed as the destination and the extend bit are subtracted from zero. The result is stored in the destination location. The size of the operation may be specified to be byte, word, or long.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set if the result is negative. Cleared otherwise.
Z — Cleared if the result is non-zero. Unchanged otherwise.
V — Set if an overflow is generated. Cleared otherwise.
C — Set if a borrow is generated. Cleared otherwise.
X — Set the same as the carry bit.

Instruction Format:



Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — byte operation.
- 01 — word operation.
- 10 — long operation.

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed.

NOP

No Operation

NOP

Operation:

Assembler Syntax: NOP

Attributes: Unsized

Description: No operation occurs. The processor state, other than the program counter, is unaffected. Execution continues with the instruction following the NOP instruction.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1

NOT — Logical Complement — NOT

Operation: ~(Destination) → Destination
NOT <ea>

Assembler Syntax: NOT <ea>

Attributes: Size = (Byte, Word, Long)

Description: The one's complement of the destination operand is taken and the result stored in the destination location. The size of the operation may be specified to be byte, word, or long.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Always cleared.
C — Always cleared.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	Size		Effective Address					

Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — byte operation.
- 01 — word operation.
- 10 — long operation.

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed.

Operation: (Destination) v (Source) →Destination
OR <ea> TO Dn
OR Dn TO <ea>

Assembler Syntax: OR <ea>,Dn
OR Dn,<ea>

Attributes: Size = (Byte, Word, Long)

Description: Inclusive OR the source operand to the destination operand, and store the result in the destination location. The size of the operation may be specified to be byte, word or long. The contents of an addresss register may not be used as an operand.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N — Set if the most significant bit of the result is set. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Always cleared.
C — Always cleared.
X — Not affected.

Instruction Format:

15	14	13	12	11 10 9	8 7 6	5 4 3 2 1 0
1	0	0	0	Register	Op-Mode	Effective Address

Instruction Fields:
Register field — Specifies any of the eight data registers.
Op-Mode field —

Byte	Word	Long	Operation
000	001	010	(<Dn>) v (<ea>) →<Dn>
100	101	110	(<ea>) v (<Dn>) →<ea>

Effective Address field —
If the location specified is a source operand then only data addressing modes are allowed.
If the location specified is a destination operand then only alterable memory addressing modes are allowed.

Operation: (Destination) v Immediate Data →Destination
OR #<data> TO <ea>

Assembler Syntax: ORI #<data>,<ea>

Attributes: Size = (Byte, Word, Long)

Description: Inclusive OR the immediate data to the destination operand, and store the result in the destination location. The size of the operation may be specified to be byte, word or long. The size of the immediate data matches the operation size. Access to the status register is available in two modes:

1. If the operation size is byte, the operation affects only the low order byte of the status register, which contains the condition codes.
2. If the operation size is word, the operation is performed on the whole status register. This is a privileged operation.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N — Set if the most significant bit of the result is set. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Always cleared.
C — Always cleared.
X — Not affected.

When accessing the status register, the condition codes are affected according to the operation.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	Size		Effective Address					
Word Data (16 bits)								Byte Data (8 bits)							
Long Data (32 bits, including previous word)															

Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — byte operation.
- 01 — word operation.
- 10 — long operation.

Effective Address field — Specifies the destination operand. Only data alterable or the status register addressing modes are allowed.

Immediate field — (Data immediately following the instruction):
- If size = 00, then the data is the low order byte of the immediate word.
- If size = 01, then the data is the entire immediate word
- If size = 10, then the data is the next two immediate words.

Operation: Destination →SP@-
PEA <ea>

Assembler Syntax: PEA <ea>

Attributes: Size = (Long)

Description: The effective address is computed and pushed onto the stack. A long word address is pushed onto the stack.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	Effective Address					

Instruction Fields:
Effective Address field — Specifies the address to be pushed onto the stack. Only control addressing modes are allowed.

RESET — Reset External Devices — RESET

Operation:

Assembler Syntax: RESET

Attributes: Unsized

Description: The reset line is asserted causing all external devices to be reset. The processor state, other than the program counter, is unaffected and execution continues with the next instruction.

Note: This is a privileged instruction.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0

ROL ROR

Rotate (without Extend)

ROL ROR

Operation: (Destination) Rotated by <count> → Destination

Assembler Syntax:
ROd Dy BY Dx
ROd Dy #<data>
ROd <ea>
ROd Dx, Dy
ROd #<data>,Dy
ROd <ea>

Attributes: Size = (Byte, Word, Long)

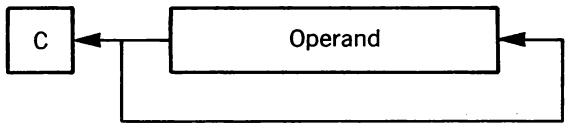
Description: Rotate the bits of the operand in the direction specified. The extend bit is not included in the rotation. The shift count for the rotation of a register may be specified in two different ways:

1. Immediate — the shift count is specified in the instruction (shift range, 1-8).
2. Register — the shift count is contained in a data register specified in the instruction.

The size of the operation may be specified to be byte, word or long. The content of memory may be rotated one bit only where the operand size is restricted to a word.

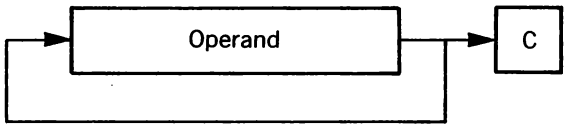
For ROL, the operand is rotated left; the number of positions shifted is the shift count. Bits shifted out of the high order bit go both to the carry bit and back into the low order bit. The extend bit is not modified or used.

ROL:



For ROR, the operand is rotated right; the number of positions shifted is the shift count. Bits shifted out of the low order bit go both to the carry bit and back into the high order bit. The extend bit is not modified or used.

ROR:

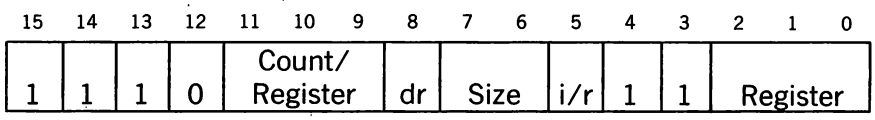


Condition Codes:

X	N	Z	V	C
—	*	*	0	*

N — Set if the most significant bit of the result is set. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Always cleared.
C — Set according to the last bit shifted out of the operand. Cleared for a shift count of zero.
X — Not affected.

Instruction Format (Register Rotate):



Instruction Fields (Register Rotate):

Count/Register field —
If i/r = 0, the rotate count is specified in this field. The values 0, 1-7 represent a range of 8, 1 to 7 respectively.
If i/r = 1, the rotate count (modulo 64) is contained in the data register specified in this field.

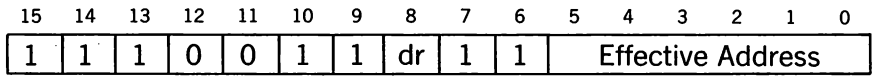
dr field — Specifies the direction of the rotate:
0 — rotate right.
1 — rotate left.

Size field — Specifies the size of the operation:
00 — byte operation.
01 — word operation.
10 — long operation.

i/r field —
If i/r = 0, specifies immediate rotate count.
If i/r = 1, specifies register rotate count.

Register field — Specifies a data register whose content is to be rotated.

Instruction Format (Memory Rotate):



Instruction Fields (Memory Rotate):

dr field — Specifies the direction of the rotate:
0 — rotate right.
1 — rotate left.

Effective Address field — Specifies the operand to be rotated. Only memory alterable addressing modes are allowed.

ROXL ROXR

Rotate with Extend

ROXL ROXR

Operation: (Destination) Rotated by <count> →Destination
ROXd Dy BY Dx
ROXd Dy BY #<data>
ROXd <ea>

Assembler Syntax:
ROXd Dx,Dy
ROXd #<data>,Dy
ROXd <ea>

Attributes: Size = (Byte, Word, Long)

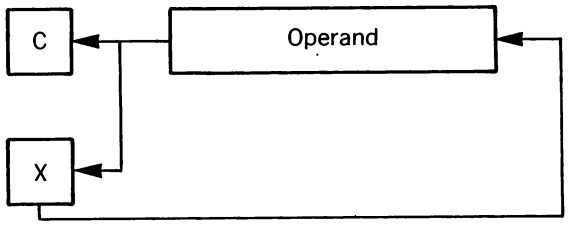
Description: Rotate the bits of the destination operand in the direction specified. The extend bit is included in the rotation. The shift count for the rotation of a register may be specified in two different ways:

1. Immediate — the shift count is specified in the instruction (shift range, 1-8).
2. Register — the shift count is contained in a data register specified in the instruction.

The size of the operation may be specified to be byte, word or long. The content of memory may be rotated one bit only where the operand size is restricted to a word.

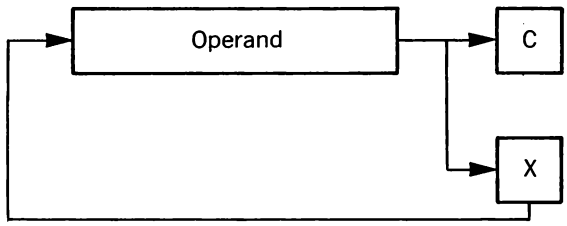
For ROXL, the operand is rotated left; the number of positions shifted is the shift count. Bits shifted out of the high order bit go to both the carry and extend bits; the previous value of the extend bit is shifted into the low order bit.

ROXL:



For ROXR, the operand is rotated right; the number of positions shifted is the shift count. Bits shifted out of the low order bit go to both the carry and extend bits; the previous value of the extend bit is shifted into the high order bit.

ROXR:



ROXL ROXR

Rotate with Extend

ROXL ROXR

Condition Codes:

X	N	Z	V	C
*	*	*	0	*

N — Set if the most significant bit of the result is set. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Always cleared.
C — Set according to the last bit shifted out of the operand. Set to the value of the extend bit for a shift count of zero.
X — Set according to the last bit shifted out of the operand. Unaffected for a shift count of zero.

Instruction Format (Register Rotate):

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	Count/ Register	dr	Size	i/r	1	0	Register

Instruction Fields (Register Rotate):

Count/Register field:
If i/r = 0, the rotate count is specified in this field. The values 0, 1-7 represent range of 8, 1 to 7 respectively.
If i/r = 1, the rotate count (modulo 64) is contained in the data register specified in this field.

dr field — Specifies the direction of the rotate:
0 — rotate right.
1 — rotate left.

Size field — Specifies the size of the operation:
00 — byte operation.
01 — word operation.
10 — long operation.

i/r field —
If i/r = 0, specifies immediate rotate count.
If i/r = 1, specifies register rotate count.

Register field — Specifies a data register whose content is to be rotated.

Instruction Format (Memory Rotate):

15	14	13	12	11	10	9	8	7	6	5 4 3 2 1 0
1	1	1	0	0	1	0	dr	1	1	Effective Address

Instruction Fields (Memory Rotate):

dr field — Specifies the direction of the rotate:
0 — rotate right.
1 — rotate left.

Effective Address field — Specifies the operand to be rotated. Only memory alterable addressing modes are allowed.

Operation: SP@+ →SR; SP@+ →PC
RTE

Assembler Syntax: RTE

Attributes: Unsized

Description: The status register and program counter are pulled from the system stack. The previous status register and program counter are lost. All bits in the status register are affected.

Note: This is a privileged instruction.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set according to the content of the word on the stack.
Z — Set according to the content of the word on the stack.
V — Set according to the content of the word on the stack.
C — Set according to the content of the word on the stack.
X — Set according to the content of the word on the stack.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1

Operation: SP@+ →CC; SP@+ →PC
RTR

Assembler Syntax: RTR

Attributes: Unsized

Description: The condition codes and program counter are pulled from the stack. The previous condition codes and program counter are lost. The supervisor portion of the status register is unaffected.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set according to the content of the word on the stack.
Z — Set according to the content of the word on the stack.
V — Set according to the content of the word on the stack.
C — Set according to the content of the word on the stack.
X — Set according to the content of the word on the stack.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	1

Operation: SP@+ → PC
RTS

Assembler Syntax: RTS

Attributes: Unsized

Description: The program counter is pulled from the stack. The previous program counter is lost.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1

Operation: $(Destination)_{10} - (Source)_{10} - X \rightarrow Destination$
SBCD Dy FROM Dx
SBCD Ay@- FROM Ax@-

Assembler Syntax: SBCD Dy,Dx
SBCD -(Ay),-(Ax)

Attributes: Size = (Byte)

Description: Subtract the source operand from the destination operand along with the extend bit, and store the result in the destination location. The subtraction is performed using binary coded decimal arithmetic. The operands may be addressed in two different ways:

1. Data register to data register: The operands are contained in the data registers specified in the instruction.
2. Memory to memory: The operands are addressed with the predecrement addressing mode using the address registers specified in the instruction.

This operation is a byte operation only.

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

N — Undefined
Z — Cleared if the result is non-zero. Unchanged otherwise.
V — Undefined.
C — Set if a borrow (decimal) is generated. Cleared otherwise.
X — Set the same as the carry bit.

Instruction Format:

15	14	13	12	11 10 9	8	7	6	5	4	3	2 1 0
1	0	0	0	Register Rx	1	0	0	0	0	R/M	Register Ry

Instruction Fields:

Register Rx field — Specifies the destination register:
If R/M = 0, specifies a data register.
If R/M = 1, specifies an address register for the predecrement addressing mode.

R/M field — Specifies the operand addressing mode:
0 — The operation is data register to data register.
1 — The operation is memory to memory.

Register Ry field — Specifies the source register:
If R/M = 0, specifies a data register.
If R/M = 1, specifies an address register for the predecrement addressing mode.

Operation: If cc then 1's → Destination else 0's → Destination
Scc <ea>

Assembler Syntax: Scc <ea>

Attributes: Size = (Byte)

Description: The specified condition code is tested; if the condition is true, the byte specified by the effective address is set to TRUE (all ones), otherwise that byte is set to FALSE (all zeroes). "cc" may specify the following conditions:

CC — carry clear	LS — low or same
CS — carry set	LT — less
EQ — equal	MI — minus
F — never true	NE — not equal
GE — greater or equal	PL — plus
GT — greater	T — always true
HI — high	VC — no overflow
LE — less or equal	VS — overflow

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11 10 9 8	7	6	5 4 3 2 1 0
0	1	0	1	Condition	1	1	Effective Address

Instruction Fields:

Condition field — One of sixteen conditions:

CC — 0100	LS — 0011
CS — 0101	LT — 1101
EQ — 0111	MI — 1011
F — 0001	NE — 0110
GE — 1100	PL — 1010
GT — 1110	T — 0000
HI — 0010	VC — 1000
LE — 1111	VS — 1001

Effective Address field — Specifies the location in which the true/false byte is to be stored. Only data alterable addressing modes are allowed.

STOP Load Status Register and Stop STOP

Operation: Immediate Data → SR; STOP
STOP #xxx

Assembler Syntax: STOP #xxx

Attributes: Unsized

Description: The immediate operand is moved into the entire status register; the program counter is advanced to point to the next instruction, and the processor stops fetching and executing instructions. Execution of instructions resumes when a trace, interrupt, or reset exception occurs. A trace exception will occur if the trace state is on when the STOP instruction is executed. If an interrupt request arrives whose priority is higher than the current processor priority, an interrupt exception occurs, otherwise the interrupt request has no effect. If the bit of the immediate data corresponding the the S-bit is off, execution of the instruction will cause a privilege violation. External reset will always initiate reset exception processing.
Note: This is a privilege instruction.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set according to the immediate operand.
Z — Set according to the immediate operand.
V — Set according to the immediate operand.
C — Set according to the immediate operand.
X — Set according to the immediate operand.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	0
Immediate Data															

Instruction Fields:
Immediate field — Specifies the data to be loaded into the status register.

Operation: (Destination) - (Source) → Destination
SUB <ea> FROM Dn
SUB Dn FROM <ea>

Assembler Syntax: SUB <ea>,Dn
SUB Dn,<ea>

Attributes: Size = (Byte, Word, Long)

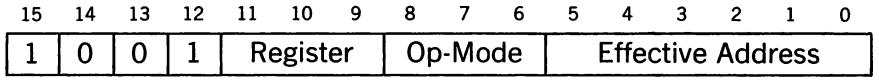
Description: Subtract the source operand from the destination operand and store the result in the destination. The size of the operation may be specified to be byte, word or long. The mode of the instruction indicates which operand is the source and which is the destination as well as the operand size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Set if an overflow is generated. Cleared otherwise.
C — Set if a borrow is generated. Cleared otherwise.
X — Set the same as the carry bit.

Instruction Format:



Instruction Fields:

Register field — Specifies any of the eight data registers.

Op-Mode field —

Byte	Word	Long	Operation
000	001	010	(<Dn>) - (<ea> → <Dn>
100	101	110	(<ea>) - (<Dn>) → <ea>

Effective Address field — Determines addressing mode:

If the location specified is a source operand, then all addressing modes are allowed. If the operation size is byte, then the address register direct addressing mode is not allowed.

If the location specified is a destination operand, then only alterable memory addressing modes are allowed.

Operation: (Destination) - (Source) → Destination
SUB <ea> FROM An

Assembler Syntax: SUBA <ea>,An

Attributes: Size = (Word, Long)

Description: Subtract the source operand from the destination address register, and store the result in the address register. The size of the operation may be specified to be word or long.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11 10 9	8 7 6	5 4 3 2 1 0
1	0	0	1	Register	Op-Mode	Effective Address

Instruction Fields:

Register field — Specifies any of the eight address registers. This is always the destination.

Op-Mode field — Specifies the size of the operation:

011 — Word operation. The source operand is sign-extended to a long operand and the operation is performed on the address register using all 32 bits.

111 — Long operations.

Effective Address field — Specifies the source operand. All addressing modes are allowed.

Operation: (Destination) - Immediate Data → Destination
SUB #<data> FROM <ea>

Assembler Syntax: SUBI #<data>,<ea>

Attributes: Size = (Byte, Word, Long)

Description: Subtract the immediate data from the destination operand, and store the result in the destination location. The size of the operation may be specified to be byte, word or long. The size of the immediate data matches the operation size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Set if an overflow is generated. Cleared otherwise.
C — Set if a borrow is generated. Cleared otherwise.
X — Set the same as the carry bit.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	Size		Effective Address					
Word Data (16 bits)								Byte Data (8 bits)							
Long Data (32 bits, including previous word)															

Instruction Fields:
Size field — Specifies the size of the operation.
00 — byte operation.
01 — word operation.
10 — long operation.
Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed.
Immediate field — (Data immediately following the instruction)
If size = 00, then the data is the low order byte of the immediate word.
If size = 01, then the data is the entire immediate word.
If size = 10, then the data is the next two immediate words.

Operation: (Destination) - Immediate Data →Destination
SUBQ #<data> FROM <ea>

Assembler Syntax: SUBQ #<data>,<ea>

Attributes: Size = (Byte, Word, Long)

Description: Subtract the immediate data from the destination operand. The data range is from 1-8. The size of the operation may be specified to be byte, word, or long. Word and long operations are also allowed on the address registers.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set if the result is negative. Cleared otherwise.
Z — Set if the result is zero. Cleared otherwise.
V — Set if an overflow is generated. Cleared otherwise.
C — Set if a borrow is generated. Cleared otherwise.
X — Set the same as the carry bit.
The condition codes are not affected if a subtraction from an address register is made.

Instruction Format:

15	14	13	12	11 10 9	8	7 6	5 4 3 2 1 0
0	1	0	1	Data	1	Size	Effective Address

Instruction Fields:
Data field — Three bits of immediate data, 0, 1-7 representing a range of 8, 1 to 7 respectively.
Size field — Specifies the size of the operation:
00 — byte operation.
01 — word operation.
10 — long operation.
Effective Address field — Specifies the destination location. Only alterable addressing modes are allowed. If the operation size is byte, the address register direct addressing mode is not allowed.

Operation: (Destination) - (Source) - X →Destination
SUBX Dy FROM Dx
SUBX Ay@- FROM Ax@-

Assembler Syntax: SUBX Dy,Dx
SUBX -(Ay),-(Ax)

Attributes: Size = (Byte, Word, Long)

Description: Subtract the source operand from the destination operand along with the extend bit, and store the result in the destination location. The operands may be addressed in two different ways:

1. Data register to data register: The operands are contained in data registers specified in the instruction.
2. Memory to memory. The operands are contained in memory and addressed with the predecrement addressing mode using the address registers specified in the instruction.

The size of the operation may be specified to be byte, word or long.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

N — Set if the result is negative. Cleared otherwise.
Z — Cleared if the result is non-zero. Unchanged otherwise.
V — Set if an overflow is generated. Cleared otherwise.
C — Set if a carry is generated. Cleared otherwise.
X — Set the same as the carry bit.

Instruction Format:

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	0	0	1	Register Rx	1	Size	0	0	R/M	Register Ry

Instruction Fields:

Register Rx field — Specifies the destination register:

If R/M = 0, specifies a data register.

If R/M = 1, specifies an address register for the predecrement addressing mode.

Size field — Specifies the size of the operation:

00 — byte operation.

01 — word operation.

10 — long operation.

R/M field — Specifies the operand addressing mode:

0 — The operation is data register to data register.

1 — The operation is memory to memory.

Register Ry field — Specifies the source register:

If R/M = 0, specifies a data register.

If R/M = 1, specifies an address register for the predecrement addressing mode.

SWAP — Swap Register Halves — SWAP

Operation: Register [31:16]↔Register [15:0]
SWAP Dn

Assembler Syntax: SWAP Dn

Attributes: Size = (Word)

Description: Exchange the 16-bit halves of a data register.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N — Set if the most significant bit of the 32-bit result is set. Cleared otherwise.
Z — Set if the 32-bit result is zero. Cleared otherwise.
V — Always cleared.
C — Always cleared.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	0	Register		

Instruction Fields:
Register field — Specifies the data register to swap.

Operation: (Destination) Tested →CC; 1 →# 7 OF Destination
TAS <ea>

Assembler Syntax: TAS <ea>

Attributes: Size = (Byte)

Description: Test and set the byte operand addressed by the effective address field. The current value of the operand is tested and N and Z are set accordingly. The high order bit of the operand is set. The operation is indivisible (using a read-modify-write memory cycle) to allow synchronization of several processors.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N — Set if the most significant bit of the operand is set. Cleared otherwise.
Z — Set if the operand is zero. Cleared otherwise.
V — Always cleared.
C — Always cleared.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	Effective Address					

Instruction Fields:

Effective Address field — Specifies the location of the tested operand. Only data alterable addressing modes are allowed.

Operation: PC → SSP@–; SR → SSP@–; (Vector) → PC
TRAP #<vector>

Assembler Syntax: TRAP #<vector>

Attributes: Unsized

Description: The processor initiates exception processing. The vector number is generated to reference the TRAP instruction exception vector specified by the low order four bits of the instruction. Sixteen TRAP instruction vectors are available.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	0	Vector			

Instruction Fields:

Vector field — Specifies which trap vector contains the new program counter to be loaded.

TRAPV Trap on Overflow TRAPV

Operation: If V then TRAP
TRAPV

Assembler Syntax: TRAPV

Attributes: Unsized

Description: If the overflow condition is on, the processor initiates exception processing. The vector number is generated to reference the TRAPV exception vector. If the overflow condition is off, no operation is performed, and execution continues with the next instruction in sequence.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	0

Operation: (Destination) Tested →CC
TST <ea>

Assembler Syntax: TST <ea>

Attributes: Size = (Byte, Word, Long)

Description: Compare the operand with zero. No results are saved, however, the condition codes are set according to results of the test. The size of the operation may be specified to be byte, word or long.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

N — Set if the operand is negative. Cleared otherwise.
Z — Set if the operand is zero. Cleared otherwise.
V — Always cleared.
C — Always cleared.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	Size		Effective Address					

Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — byte operation.
- 01 — word operation.
- 10 — long operation.

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed.

Operation: An → SP; SP@+ → An
UNLK An

Assembler Syntax: UNLK An

Attributes: Unsized

Description: The stack pointer is loaded from the specified address register. The address register is then loaded with the long word pulled from the top of the stack.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

N — Not affected.
Z — Not affected.
V — Not affected.
C — Not affected.
X — Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	1	Register		

Instruction Fields:
Register field — Specifies the address register through which the unlinking is to be done.

APPENDIX C
INSTRUCTION FORMAT SUMMARY

C.0 INTRODUCTION

This appendix provides a summary of the first word in each instruction of the instruction set. Table C-1 is an operation code (op-code) map which illustrates how bits 15 through 12 are used to specify the operations. The remaining paragraph groups the instructions according to the op-code map.

Table C-1. Operation Code Map

Bits 15 thru 12	Operation
0000	Bit Manipulation/MOVEP/Immediate
0001	Move Byte
0010	Move Long
0011	Move Word
0100	Miscellaneous
0101	ADDQ/SUBQ/S_{CC}/ DB_{CC}
0110	B_{CC}
0111	MOVEQ
1000	OR/DIV/SBCD
1001	SUB/SUBX
1010	(Unassigned)
1011	CMP/EOR
1100	AND/MUL/ABCD/EXG
1101	ADD/ADDX
1110	Shift/Rotate
1111	(Unassigned)

C.1 BIT MANIPULATION, MOVE PERIPHERAL, IMMEDIATE INSTRUCTIONS

Dynamic Bit

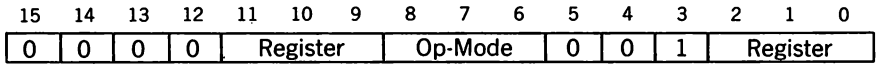
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Register			1	Type		Effective Address					

Static Bit

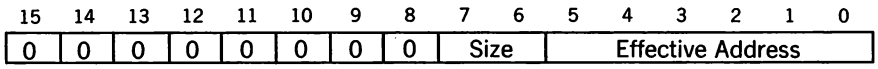
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	Type		Effective Address					

Bit Type Codes: TST = 00, CHG = 01, CLR = 10, SET = 11

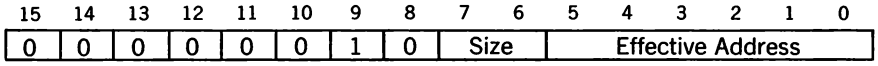
MOVEP



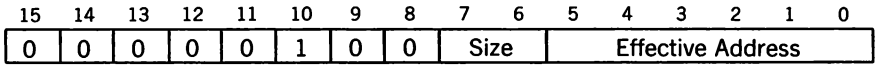
OR Immediate



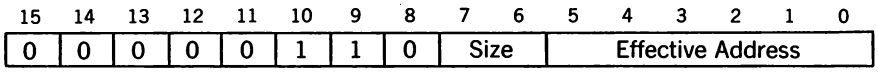
AND Immediate



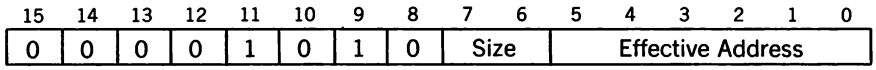
SUB Immediate



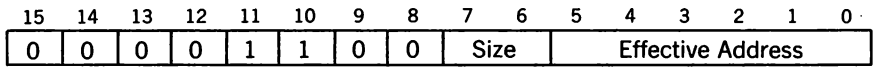
ADD Immediate



EOR Immediate

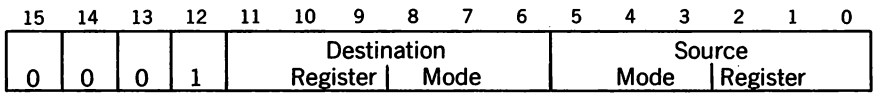


CMP Immediate



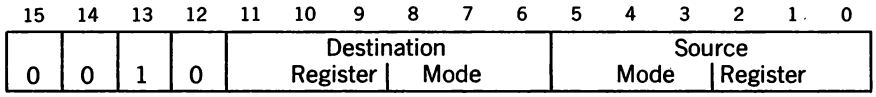
C.2 MOVE BYTE INSTRUCTION

MOVE Byte



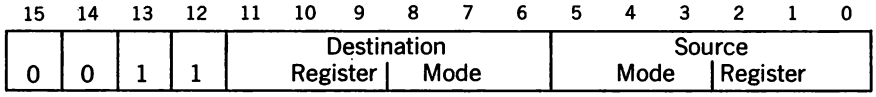
C.3 MOVE LONG INSTRUCTION

MOVE Long



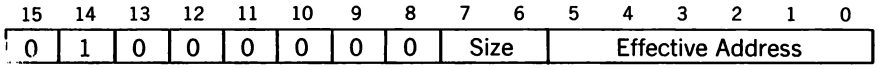
C.4 MOVE WORD INSTRUCTION

MOVE Word

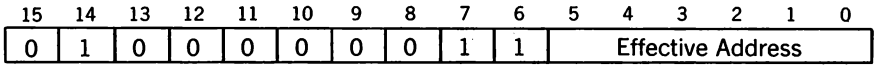


C.5 MISCELLANEOUS INSTRUCTIONS

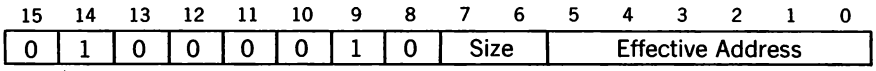
NEGX



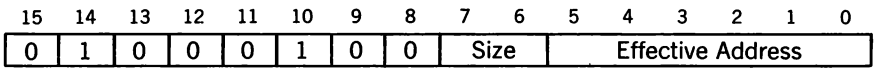
MOVE from SR



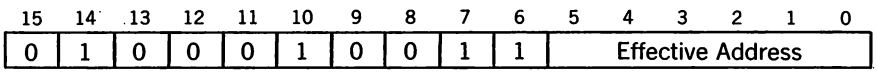
CLR



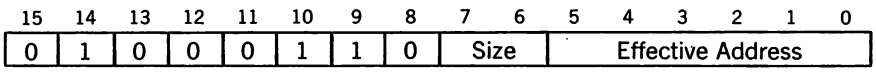
NEG



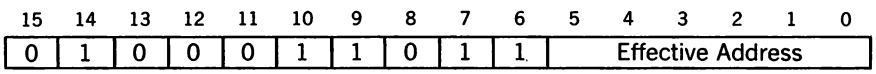
MOVE to CCR



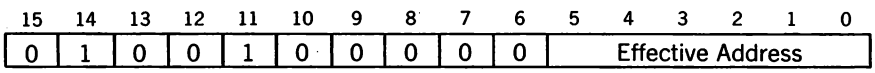
NOT



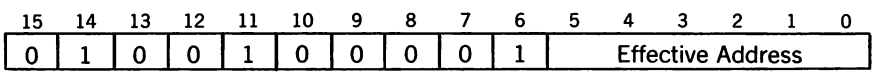
MOVE to SR



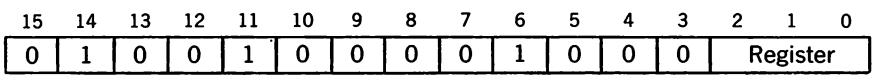
NBCD



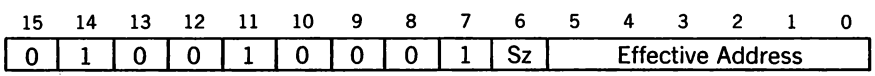
PEA



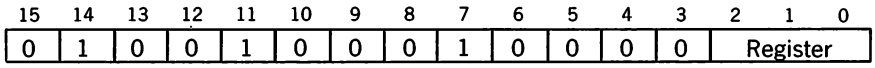
SWAP



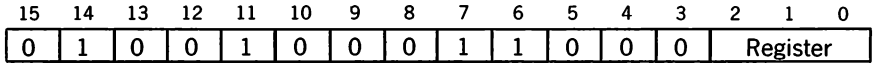
MOVEM Registers to EA



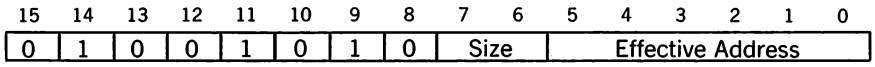
EXTW



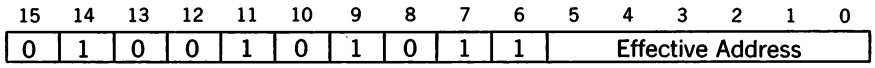
EXTL



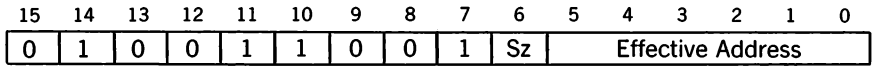
TST



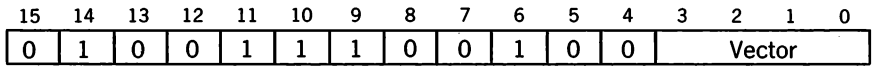
TAS



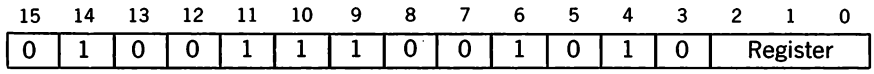
MOVEM EA to Registers



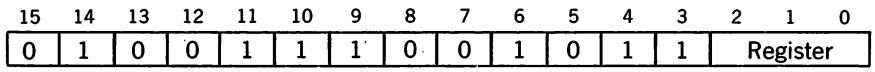
TRAP



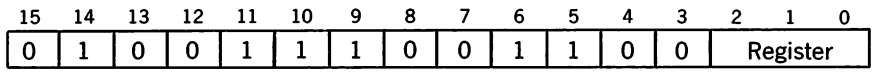
LINK



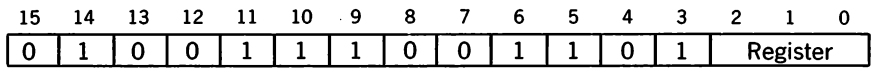
UNLK



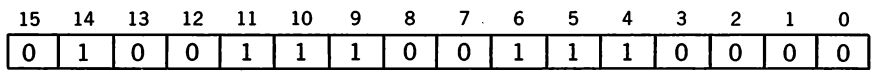
MOVE to USP



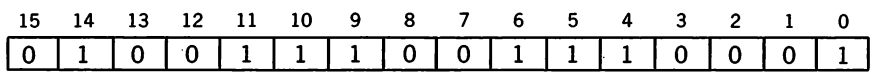
MOVE from USP



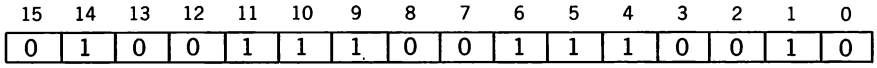
RESET



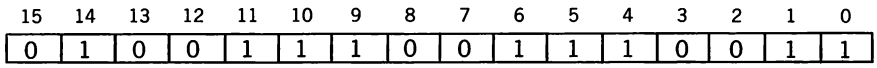
NOP



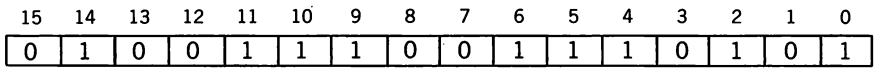
STOP



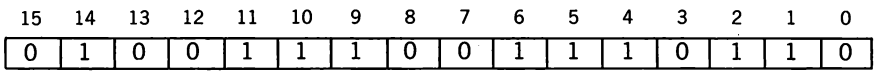
RTE



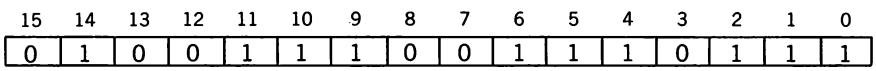
RTS



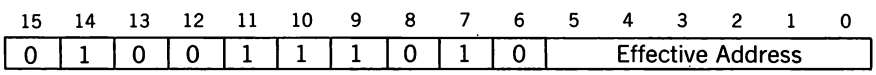
TRAPV



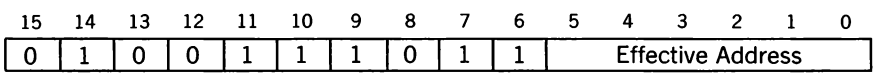
RTR



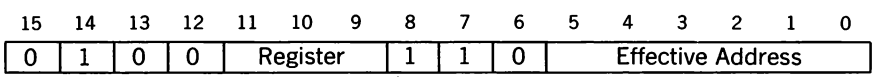
JSR



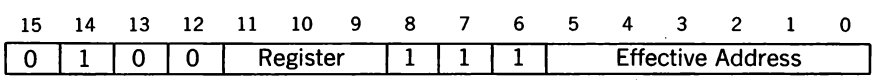
JMP



CHK

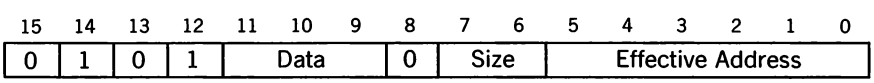


LEA

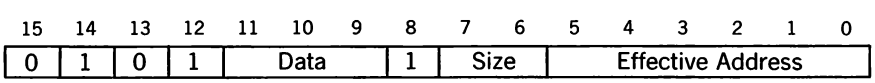


C.6 ADD QUICK, SUBTRACT QUICK, SET CONDITIONALLY, DECREMENT INSTRUCTIONS

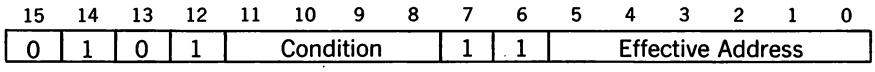
ADDQ



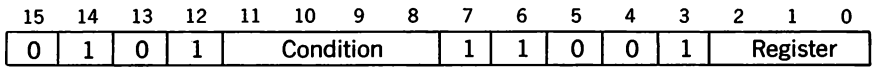
SUBQ



Scc

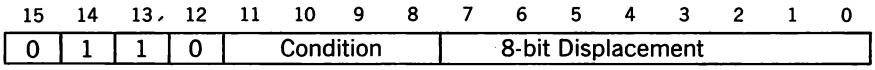


DBcc



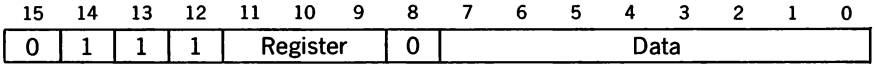
C.7 BRANCH CONDITIONALLY INSTRUCTION

Bcc



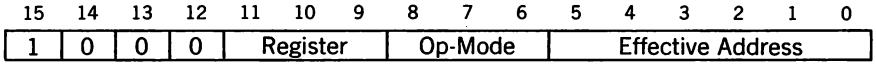
C.8 MOVE QUICK INSTRUCTION

MOVEQ

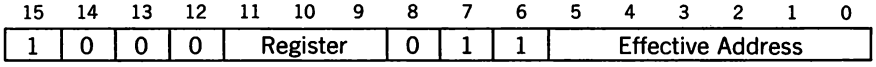


C.9 OR, DIVIDE, SUBTRACT DECIMAL INSTRUCTIONS

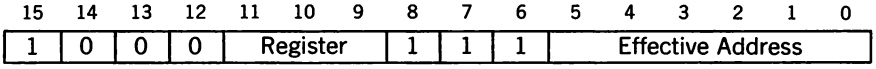
OR



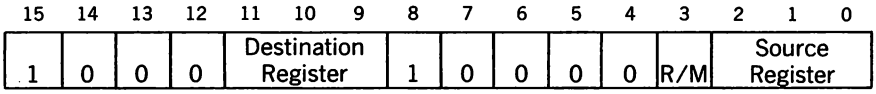
DIVU



DIVS



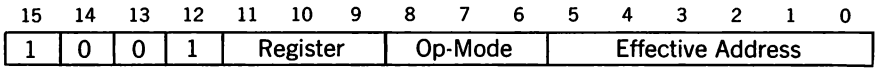
SBCD



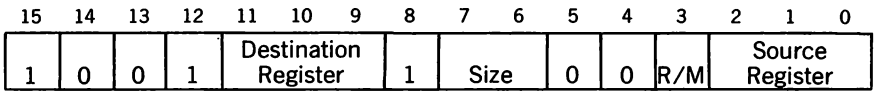
R/M (register/memory): register - register = 0, memory - memory = 1

C.10 SUBTRACT, SUBTRACT EXTENDED INSTRUCTIONS

SUB



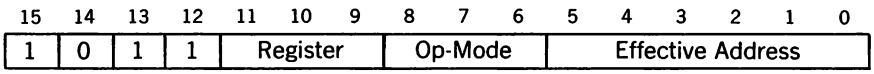
SUBX



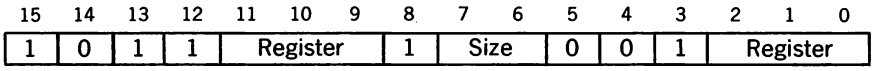
R/M (register/memory): register- register = 0, memory - memory = 1

C.11 COMPARE, EXCLUSIVE OR INSTRUCTIONS

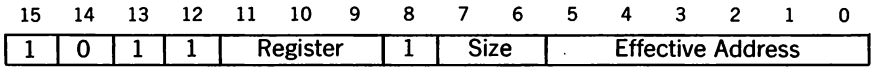
CMP



CMPM

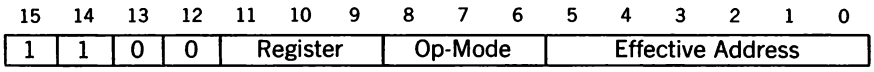


EOR

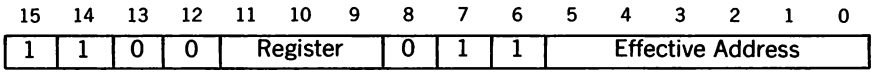


C.12 AND, MULTIPLY, ADD DECIMAL, EXCHANGE INSTRUCTIONS

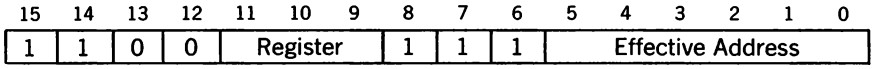
AND



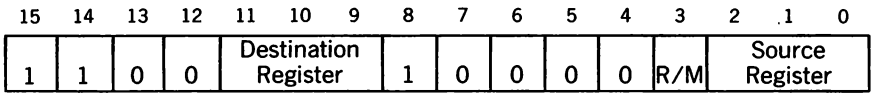
MULU



MULS

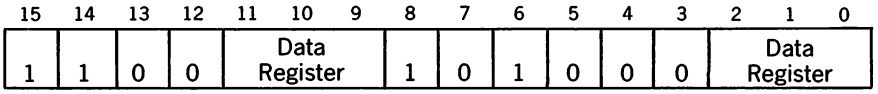


ABCD

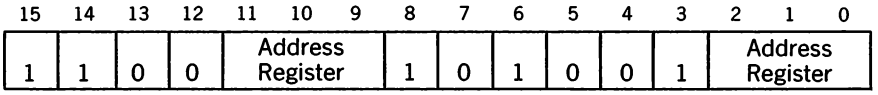


R/M (register/memory): register - register = 0, memory - memory = 1

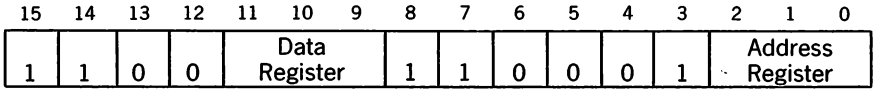
EXGD



EXGA

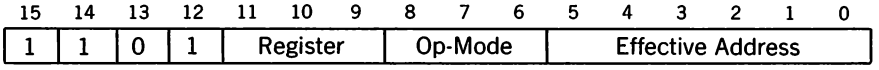


EXGM

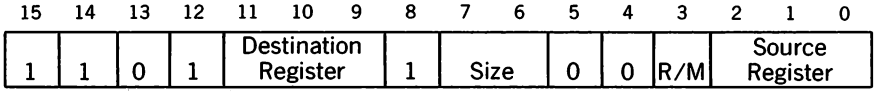


C.13 ADD, ADD EXTENDED INSTRUCTIONS

ADD



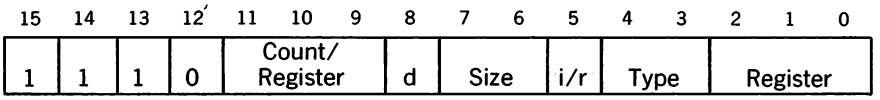
ADDX



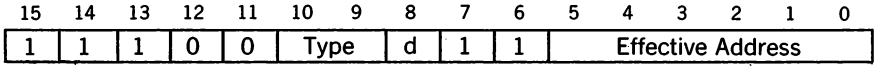
R/M (register/memory): register - register = 0, memory - memory = 1

C.14 SHIFT/ROTATE INSTRUCTIONS

Data Register Shifts



Memory Shifts



Shift Type Codes: AS = 00,LS = 01, ROX = 10, RO = 11
d (direction): Right = 0, Left = 1
i/r (count source): Immediate Count = 0, Register Count = 1

APPENDIX D
INSTRUCTION EXECUTION TIMES

D.0 INTRODUCTION

This appendix contains listings of the instruction execution times in terms of the number of internal cycles required for an instruction to execute. One internal cycle is equal to two cycles of the processor clock input. It is also assumed that the memory cycle time is no greater than four cycles of the processor clock input to prevent the introduction of wait states in the bus cycle.

These timings are for the entire instruction, including operand fetches, operand writes, and instruction reads.

D.1 EFFECTIVE ADDRESS OPERAND CALCULATION TIMING

Table D-1 gives the overhead time required for processing an effective address operand. The time includes fetching of any extension words, the address computation, and the fetching of the memory operand.

Table D-1. Effective Address Operand Calculation Timing

Addressing Mode		Time	
		Byte, Word	Long
	Register		
Dn	Data Register Direct	0	0
An	Address Register Direct	0	0
	Memory		
An@	Address Register Indirect	2	4
An@+	Address Register Indirect with Postincrement	2	4
An@-	Address Register Indirect with Predecrement	3	5
An@(d)	Address Register Indirect with Displacement	4	6
An@(d, ix)	Address Register Indirect with Index	5	7
xxx.W	Absolute Short	4	6
xxx.L	Absolute Long	6	8
PC@(d)	Program Counter with Displacement	4	6
PC@(d, ix)	Program Counter with Index	5	7
#xxx	Immediate	2	4

D-2 MOVE INSTRUCTION TIMING

Tables D-2 and D-3 give the timing for the move instruction. These are complete instruction timings, including operand reads and writes, and instruction reads.

Table D-2. Move Byte and Word Instruction Timing

Source	Destination								
	Dn	An	An@	An@+	An@-	An@(d)	An@(d, ix)	xxx.W	xxx.L
Dn	2	2	4	4	4	6	7	6	8
An	2	2	4	4	4	6	7	6	8
An@	4	4	6	6	6	8	9	8	10
An@+	4	4	6	6	6	8	9	8	10
An@-	5	5	7	7	7	9	10	9	11
An@(d)	6	6	8	8	8	10	11	10	12
An@(d, ix)	7	7	9	9	9	11	12	11	13
xxx.W	6	6	8	8	8	10	11	10	12
xxx.L	8	8	10	10	10	12	13	12	14
PC@(d)	6	6	8	8	8	10	11	10	12
PC@(d, ix)	7	7	9	9	9	11	12	11	13
#xxx	4	4	6	6	6	8	9	8	10

Note:
The move byte and word instructions each require one memory access for operand read and operand write.

Table D-3. Move Long Instruction Timing

Source	Destination								
	Dn	An	An@	An@+	An@-	An@(d)	An@(d, ix)	xxx.W	xxx.L
Dn	2	2	6	6	6	8	9	8	10
An	2	2	6	6	6	8	9	8	10
An@	6	6	10	10	10	12	13	12	14
An@+	6	6	10	10	10	12	13	12	14
An@-	7	7	11	11	11	13	14	13	15
An@(d)	8	8	12	12	12	14	15	14	16
An@(d, ix)	9	9	13	13	13	15	16	15	17
xxx.W	8	8	12	12	12	14	15	14	16
xxx.L	10	10	14	14	14	16	17	16	18
PC@(d)	8	8	12	12	12	14	15	14	16
PC@(d, ix)	9	9	13	13	13	15	16	15	17
#xxx	6	6	10	10	10	12	13	12	14

Note:
The move long instruction requires two memory accesses each for operand read and operand write.

D.3 STANDARD INSTRUCTION TIMING

The times shown in table D-4 include the time to perform the operation, store the results, and read the next instruction. Address calculation and effective address fetch time must be added where indicated. In table D-4, the headings have the following meanings: A = address register operand, D = data register operand, E = an operand specified by an effective address, and M = memory operand.

Table D-4. Standard Instruction Timing

Instruction	Size	A op E →A	D op E →D	M op D →M
ADD	Byte, Word Long	4+ 3+#	2+ 3+#	4+ 6+
AND	Byte, Word Long		2+ 3+#	4+ 6+
CMP	Byte, Word Long	3+ 3+	2+ 3+	
DIVS			79+*	
DIVU			70+*	
EOR	Byte, Word Long		2 3+	4+ 5+
MULS			35+*	
MULU			35+*	
OR	Byte, Word Long		2+ 3+#	4+ 6+
SUB	Byte, Word Long	4+ 3+#	2+ 3+#	4+ 6+

\+ add effective address calculation time
\# 4 if the effective address is register direct
* indicates maximum value

D.4 IMMEDIATE INSTRUCTION TIMING

The times shown in table D-5 include the time to fetch immediate operands, perform the operations, store the results, and read the next operation. In table D-5, the headings have the following meanings: D = data register operand, I = immediate operand, M = memory operand, and SR = status register.

Table D-5. Immediate Instruction Timing

Instruction	Size	D op I →D	M op I →M	SR op I →SR
ADDI	Byte, Word Long	4 8	6+ 10+	
ADDQ	Byte, Word Long	2 4	4+ 6+	
ANDI	Byte, Word Long	4 8	6+ 10+	10
CMPI	Byte, Word Long	4 7	4+ 6+	
EORI	Byte, Word Long	4 8	6+ 10+	10
LDQ	Long	2	—	
ORI	Byte, Word Long	4 8	6+ 10+	10
SUBI	Byte, Word Long	4 8	6+ 10+	
SUBQ	Byte, Word Long	2 4	4+ 6+	

+ add effective address calculation time

D.5 SINGLE OPERAND INSTRUCTION TIMING

Table D-6 gives the timing for single operand instructions.

Table D-6. Single Operand Instruction Timing

Instruction	Size	Register	Memory
CLR	Byte, Word Long	2 3	4+ 6+
NBCD	Byte	3	4+
NEG	Byte, Word Long	2 3	4+ 6+
NEGX	Byte, Word Long	2 3	4+ 6+
NOT	Byte, Word Long	2 3	4+ 6+
Scc	Byte, False Byte, True	2 3	4+ 4+
TAS	Byte	2	5+
TST	Byte, Word Long	2 2	2+ 2+

+ add effective address calculation time

D.6 SHIFT/ROTATE INSTRUCTION TIMING

Table D-7 gives the timing for shift and rotate instructions.

Table D-7. Shift and Rotate Instruction Timing

Instruction	Size	Register*	Memory
ASR, ASL	Byte, Word Long	3 + n 4 + n	4+
LSR, LSL	Byte, Word Long	3 + n 4 + n	4+
ROR, ROL	Byte, Word Long	3 + n 4 + n	4+
ROXR, ROXL	Byte, Word Long	3 + n 4 + n	4+

+ add the effective address calculation time
*n = the shift count

D.7 BIT MANIPULATION INSTRUCTION TIMING

Table D-8 gives the timing for the bit manipulation instructions.

Table D-8. Bit Manipulation Instruction Timing

		Dynamic		Static	
Instruction	Size	Register	Memory	Register	Memory
BCHG	Word Long	— 4*	4+ —	— 6*	6+ —
BCLR	Word Long	— 5*	4+ —	— 7*	6+ —
BSET	Word Long	— 4*	4+ —	— 6*	6+ —
BTST	Word Long	— 3	2+ —	— 5	4+ —

+ add effective address calculation time.
* indicate maximum value

D.8 CONDITIONAL INSTRUCTION TIMING

Table D-9 gives the timing for the conditional instructions.

Table D-9. Conditional Instruction Timing

Instruction	Displacement	Successful (TRAP)	Unsuccessful (No TRAP)
B_{cc}	byte word	5 5	4 6
BRA	byte word	5 5	
BSR	byte word	9 9	
DB_{cc}		5	7
CHK		19*+	4+
TRAP		16	
TRAPV		16	2

+ add effective address calculation time
* indicates maximum value

D.9 JMP, JSR, LEA, PEA, MOVEM INSTRUCTION TIMING

Table D-10 gives the timing for the jump, jump to subroutine, load effective address, push effective address, and move multiple registers.

Table D-10. JMP, JSR, LEA, PEA, MOVEM Instruction Timing

Instruction	Size	An@	An@+	An@-	An@(d)	An@(d, ix)	xxx.W	xxx.L	PC@(d)	PC@(d, ix)
JMP		4	—	—	5	7	5	6	5	7
JSR		8	—	—	9	11	9	10	9	11
LEA		2	—	—	4	6	4	6	4	6
MOVEM	Word	6 + 2n	6 + 2n	—	8 + 2n	9 + 2n	8 + 2n	10 + 2n	8 + 2n	9 + 2n
M →R	Long	6 + 4n	6 + 4n	—	8 + 4n	9 + 4n	8 + 4n	10 + 4n	8 + 4n	9 + 4n
MOVEM	Word	4 + 2n	—	4 + 2n	6 + 2n	7 + 2n	6 + 2n	8 + 2n	—	—
R →M	Long	4 + 4n	—	4 + 4n	6 + 4n	7 + 4n	6 + 4n	8 + 4n	—	—
PEA		6	—	—	8	9	8	10	8	9

n = the number of registers to move

D.10 MULTI-PRECISION INSTRUCTIONS TIMING

Table D-11 gives the timings for the multi-precision instructions. These times include the time to fetch both operands, perform the operations, store the results, and read the next instructions.

Table D-11. Multi-Precision Instructions Timing

Instruction	Size	D op D →D	M op M →M
ADDX	Byte, Word Long	2 4	9 15
CMPM	Byte, Word Long		6 10
SUBX	Byte, Word Long	2 4	9 15
ABCD	Byte	3	9
SBCD	Byte	3	9

D.11 MISCELLANEOUS INSTRUCTIONS TIMING

Table D-12 gives the timing for some miscellaneous instructions.

Table D-12. Miscellaneous Instructions Timing

Instruction	Size	Register →Memory	Memory →Register	Register	Memory
MOVE from SR	—	—	—	3	4+
MOVE to CCR	—	—	—	6	6+
MOVE to SR	—	—	—	6	6+
MOVEP	Word Long	8 12	8 12	— —	— —
EXG	—	—	—	3	—
EXT	Word Long	— —	— —	2 2	— —
LINK	—	—	—	8	—
MOVE from USP	—	—	—	2	—
MOVE to USP	—	—	—	2	—
NOP	—	—	—	2	—
RESET	—	—	—	66	—
RTE	—	—	—	10	—
RTR	—	—	—	10	—
RTS	—	—	—	8	—
STOP	—	—	—	2	—
SWAP	—	—	—	2	—
UNLK	—	—	—	6	—

+ add effective address calculation time

D.12 EXCEPTION PROCESSING TIMING

Table D-13 gives the timing for exception processing. These times include all automatic stacking, the vector fetch, and the fetch of the first instruction of the handler routine.

Table D-13. Exception Processing Timing

Exception	Time
Address Error	24
Bus Error	24
Interrupt	21*
Illegal Instruction	16
Privileged Instruction	16
Trace	16

* Interrupt acknowledge is assumed to take two cycles.

APPENDIX E
PINOUT DIAGRAM

E.0 INTRODUCTION

This appendix contains a pinout diagram of the MC68000. Refer to figure E-1.

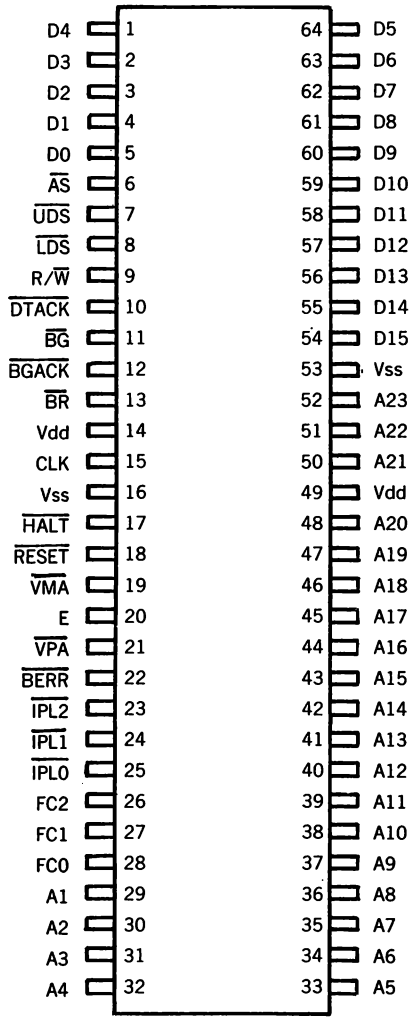


Figure E-1. Pinout Diagram

APPENDIX F
CROSS MACRO ASSEMBLER

F.0 INTRODUCTION

This appendix contains a brief description of the cross macro assembler available from Motorola. A summary of the instruction set along with the assembler syntax and the condition code results is also given.

F.1 M68000 CROSS MACRO ASSEMBLER

The M68000 Cross Macro Assembler is used on various host computer systems to translate a MC68000 source program into machine language. The present systems that the assembler can be used on are:

- M6800 EXORciser Development System
- M6809 EXORciser Development System
- IBM 370 Systems
- PDP-11

The IBM 370 and PDP-11 based assemblers can be used on other host systems with a minimum of changes. This assembler fully supports the complete MC68000 instruction set.

Programs can be developed and debugged using the MEX68KDM Design Module or the M68000 Simulator for IBM 370 Systems. The simulator and assembler also provide access to the host computers' other resources, including text editors, file systems, and peripherals. Figure F-1 is a representation of the operation of the assembler.

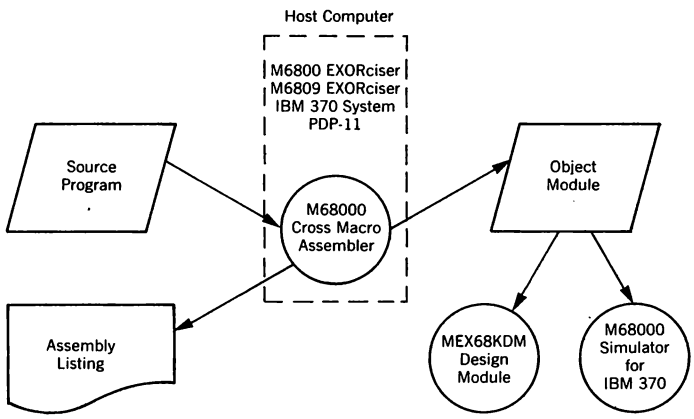


Figure F-1. M68000 Cross Macro Assembler Operation

The M68000 Cross Macro Assembler package contains:

- A FORTRAN source program on magnetic tape (IBM 370 and PDP-11 systems) or diskette (M6800 and M6809 EXORciser systems)
- Installation instructions
- The M68000 Cross Macro Assembler reference manual

The M68000 Cross Macro Assembler includes the following features:

- Absolute Code Generation
- Macro Capability
- Conditional Assembly
- Complex Expressions
- Symbol Table Listing

Assembly Language Input

- 14 Address Modes
- Assembler Directives (Pseudo-Operations)

END	End of Program
SET/EQU	Equate Symbol
ORG/RORG	Origin
PAGE	Top of Form
LIST	Default
NOLIST	
TTL	Source Program title
NOPAGE	Do Not Page Source Output
LLEN	Set Line Length on Output
PLEN	Set Number of Lines/Pages
NOOBJ	No Object Output

- Conditional Assembly Directives

IFEQ	Assemble If Equal
IFNE	Assemble If Not Equal
ENDC	End Conditional Assembly

- Macro Directives

MACRO	Define Macro
ENDM	End Macro Definition
MEXIT	End Macro Expansion
FAIL	Display Error

Assembler Output

- Object Module — EXORciser-Compatible Format (ASCII Characters)
- Assembly Listing
- Symbol Tape

F.2 INSTRUCTION SET SUMMARY

Table F-1 is a summary of the MC68000 instruction set, the M68000 Cross Macro Assembler syntax statements, and the condition code results.

Table F-1. Instruction Set Summary

Mnemonic	Operation	Assembler Syntax	Condition Codes				
			X	N	Z	V	C
ABCD	Add Decimal with Extend	ABCD Dy,Dx ABCD -(Ay),-(Ax)	*	U	*	U	*
ADD	Add Binary	ADD <ea>,Dn ADD Dn,<ea>	*	*	*	*	*
ADDA	Add Address	ADDA <ea>,An	—	—	—	—	—
ADDI	Add Immediate	ADDI #<data>,<ea>	*	*	*	*	*
ADDQ	Add Quick	ADDQ #<data>,<ea>	*	*	*	*	*
ADDX	Add Extended	ADDX Dy,Dx ADDX -(Ay),-(Ax)	*	*	*	*	*
AND	AND Logical	AND <ea>,Dn AND Dn,<ea>	—	*	*	0	0
ANDI	AND Immediate	ANDI #<data>,<ea>	—	*	*	0	0
ASL, ASR	Arithmetic Shift	ASd Dx,Dy ASd #<data>,Dy ASd<ea>	*	*	*	*	*
Bcc	Branch Conditionally	Bcc <label>	—	—	—	—	—
BCHG	Test a Bit and Change	BCHG Dn,<ea> BCHG #<data>,<ea>	—	—	*	—	—
BCLR	Test a Bit and Clear	BCLR Dn,<ea> BCLR #<data>,<ea>	—	—	*	—	—
BRA	Branch Always	BRA <label>	—	—	—	—	—
BSET	Test a Bit and Set	BSET Dn,<ea> BSET #<data>,<ea>	—	—	*	—	—
BSR	Branch to Subroutine	BSR <label>	—	—	—	—	—
BTST	Test a Bit	BTST Dn,<ea> BTST #<data>,<ea>	—	—	*	—	—
CHK	Check Register Against Bounds	CHK <ea>,Dn	—	*	U	U	U
CLR	Clear an Operand	CLR <ea>	—	0	1	0	0
CMP	Arithmetic Compare	CMP <ea>,Dn	—	*	*	*	*
CMPA	Arithmetic Compare Address	CMPA <ea>,An	—	*	*	*	*
CMPI	Compare Immediate	CMPI #<data>,<ea>	—	*	*	*	*
CMPM	Compare Memory	CMPM (Ay)+,(Ax)+	—	*	*	*	*
DBcc	Test Condition, Decrement and Branch	DBcc Dn, <label>	—	—	—	—	—
DIVS	Signed Divide	DIVS <ea>,Dn	—	*	*	*	0
DIVU	Unsigned Divide	DIVU <ea>,Dn	—	*	*	*	0

Table F-1. Instruction Set Summary (Continued)

Mnemonic	Operation	Assembler Syntax	Condition Codes				
			X	N	Z	V	C
EOR	Exclusive OR Logical	EOR Dn,<ea>	—	*	*	0	0
EORI	Exclusive OR Immediate	EORI #<data>,<ea>	—	*	*	0	0
EXG	Exchange Registers	EXG Rx,Ry	—	—	—	—	—
EXT	Sign Extend	EXT Dn	—	*	*	0	0
JMP	Jump	JMP <ea>	—	—	—	—	—
JSR	Jump to Subroutine	JSR <ea>	—	—	—	—	—
LEA	Load Effective Address	LEA <ea>,An	—	—	—	—	—
LINK	Link and Allocate	LINK An,#<displacement>	—	—	—	—	—
LSL, LSR	Logical Shift	LSd Dx,Dy LSd #<data>,Dy LSd <ea>	*	*	*	0	*
MOVE	Move Data from Source to Destination	MOVE <ea>,<ea>	—	*	*	0	0
MOVE to CCR	Move to Condition Codes	MOVE <ea>,CCR	*	*	*	*	*
MOVE to SR	Move to the Status Register	MOVE <ea>,SR	*	*	*	*	*
MOVE from SR	Move from the Status Register	MOVE SR,<ea>	—	—	—	—	—
MOVE USP	Move User Stack Pointer	MOVE USP,An MOVE An,USP	—	—	—	—	—
MOVEA	Move Address	MOVEA <ea>,An	—	—	—	—	—
MOVEM (see note)	Move Multiple Registers	MOVEM <register list>,<ea> MOVEM <ea>,<register list>	—	—	—	—	—
MOVEP	Move Peripheral Data	MOVEP Dx,d(Ay) MOVEP d(Ay),Dx	—	—	—	—	—
MOVEQ	Move Quick	MOVEQ #<data>,Dn	—	*	*	0	0
MULS	Signed Multiply	MULS <ea>,Dn	—	*	*	0	0
MULU	Unsigned Multiply	MULU <ea>,Dn	—	*	*	0	0
NBCD	Negate Decimal with Extend	NBCD <ea>	*	U	*	U	*
NEG	Two's Complement Negation	NEG <ea>	*	*	*	*	*
NEGX	Negate with Extend	NEGX <ea>	*	*	*	*	*
NOP	No Operation	NOP	—	—	—	—	—
NOT	Logical Complement	NOT <ea>	—	*	*	0	0
OR	Inclusive OR Logical	OR <ea>,Dn OR Dn,<ea>	—	*	*	0	0
ORI	Inclusive OR Immediate	ORI #<data>,<ea>	—	*	*	0	0
PEA	Push Effective Address	PEA <ea>	—	—	—	—	—

Table F-1. Instruction Set Summary (Concluded)

Mnemonic	Operation	Assembler Syntax	Condition Codes				
			X	N	Z	V	C
RESET	Reset External Devices	RESET	—	—	—	—	—
ROL, ROR	Rotate (without extend)	ROd Dx,Dy ROd #<data>,Dy ROd <ea>	—	*	*	0	*
ROXL, ROXR	Rotate with Extend	ROXd Dx,Dy ROXd #<data>,Dy ROXd <ea>	*	*	*	0	*
RTE	Return from Exception	RTE	*	*	*	*	*
RTR	Return and Restore Condition Codes	RTR	*	*	*	*	*
RTS	Return from Subroutine	RTS	—	—	—	—	—
SBCD	Subtract Decimal with Extend	SBCD Dy,Dx SBCD -(Ay),-(Ax)	*	U	*	U	*
Scc	Set according to Condition	Scc <ea>	—	—	—	—	—
STOP	Stop Program Execution	STOP #<data>	—	—	—	—	—
SUB	Subtract Binary	SUB <ea>,Dn SUB Dn,<ea>	*	*	*	*	*
SUBA	Subtract Address	SUBA <ea>,An	—	—	—	—	—
SUBI	Subtract Immediate	SUBI #<data>,<ea>	*	*	*	*	*
SUBQ	Subtract Quick	SUBQ #<data>,<ea>	*	*	*	*	*
SUBX	Subtract with Extend	SUBX Dy,Dx SUBX -(Ay),-(Ax)	*	*	*	*	*
SWAP	Swap Register Halves	SWAP Dn	—	*	*	0	0
TAS	Test and Set an Operand	TAS <ea>	—	*	*	0	0
TRAP	Trap	TRAP #<vector>	—	—	—	—	—
TRAPV	Trap on Overflow	TRAPV	—	—	—	—	—
TST	Test and Operand	TST <ea>	—	*	*	0	0
UNLK	Unlink	UNLK An	—	—	—	—	—

Note:

<register list> specifies the registers selected for transfer to or from memory.

<register list> may be:

Rn — a single register.

Rn-Rm — a range of consecutive registers with m being greater than n.

Any combination of the above separated by a slash.

EUROPEAN MOTOROLA SEMICONDUCTOR SALES OFFICES

DENMARK
Motorola A/S
Bredebovej 23
DK-2800 Lyngby
Tel. (02) 88 44 55

FRANCE
Motorola Semiconducteurs S.A.
Headquarter
15-17, avenue de Ségur
75007 Paris
Tel. 551 50 61
Sales Office
42, avenue de La Plaine-Fleurie
38240 Meylan (Grenoble)
Tel. (76) 90 22 81

WEST GERMANY
Motorola GmbH, Geschäftsbereich Halbleiter
Headquarter
Munchner Strasse 18
8043 Unterföhring
Tel. (089) 92 481
Sales Offices
Hans Bockler-Strasse 30
3012 Langenhagen - Hannover
Tel. (0511) 77 20 37
Virnsbergerstrasse 43
8500 Nurnberg
Tel. (0911) 6 57 61
Stralsunder Strasse 1
7032 Sindelfingen
Tel. (07031) 8 30 74/8 30 75
Abraham-Lincoln-Strasse 28
6200 Wiesbaden
Tel. (06121) 76 19 21

HOLLAND
Motorola B.V.
Emmalaan 41
Utrecht
Tel. (030) 51 02 07

ITALY
Motorola S.p.A.
Headquarter
Via Gio Menotti 11
20129 Milano
Tel. 738 61 41/2/3
Sales Office
Via Portanova 10
40123 Bologna - Tel. 26 69 05
Sales Office
Via Costantino Maes 68
00162 Roma - Tel. 831 47 46

NORWAY
Motorola A/B (Service Office)
Brugt. 1
Oslo 1 Tel. (02) 41 91 40

SOUTH AFRICA
Motorola South Africa (Pty) Ltd.
P.O. Box 39586
Bramley 2018
Tel. 786 11 84

SWEDEN
Motorola AB.
Virebergsvaegen 19
17140 Solna - Tel. (08) 82 02 95

SWITZERLAND
Motorola Semiconductor Products S.A.
Alte Landstrasse 101
8702 Zollikon Tel. (01) 65 56 56

UNITED KINGDOM
Motorola Ltd.
Headquarter
York House, Empire Way
Wembley Middlesex
Tel. (01) 902 88 36
Sales Office
10-12, Mount Street, Television House
Manchester M2 5WS, Lancs
Tel. (61) 833 07 31/833 07 34
Sales Office
Colvilles Road, Kelvin Estate
East Kilbride, Scotland
Tel. (3552) 3 91 01

HEADQUARTERS EUROPEAN OPERATIONS

SWITZERLAND
Motorola Inc.
Semiconductor Products Group
16, chemin de la Voie Creuse, P.O. Box 8
1211 Genève 20
Tel. (022) 33 56 07

FRANCHISED MOTOROLA SEMICONDUCTOR DISTRIBUTORS

AUSTRIA
Elbatex GmbH
Endresstrasse 54 1238 Wien
Tel. (222) 88 55 11

BELGIUM
Diode Belgium
Rue Picard 202-204 1020 Bruxelles
Tel. (02) 428 51 05

DENMARK
Distributören Interelko Aps
Hovedgaden 16 4622 Havdrup
Tel. (03) 38 57 16

FINLAND
Field Oy
Veneentekijantie 18 00210 Helsinki 21
Tel. (80) 692 25 77

FRANCE
Bellion Electronique
Zone Industrielle de Kerscao-Brest
29219 Le Relecq-Kerhuon B.P. 16
Tel. (98) 28 03 03
Celdis S.A.
53, rue Charles-Frérot 94250 Gentilly
Tel. (01) 581 00 20
Ets. F. Feutrier S.A. (Main Office)
Rue des Trois-Glorieuses
42270 St-Priest-en-Jarez (St-Etienne)
Tel. (77) 74 67 33
Ets. F. Feutrier S.A.
Avenue Laplace Zone Industrielle
13470 Carnoux
Tel. (42) 82 16 41
Feutrier Ile de France
29, rue Ledru-Rollin 92150 Suresnes
Tel. (01) 772 46 46
Ets. Gros S.A. (Main Office)
13, rue Victor-Hugo BP 63
59350 Saint-André-lez-Lille
Tel. (20) 51 21 33
Ets. Gros S.A.
14, avenue du Général-Leclerc 54000 Nancy
Tel. (83) 35 17 35
Ets. Gros S.A.
5, rue Pascal 94800 Villejuif
Tel. (01) 678 27 27
S.C.A.I.B. S.A.
80, rue d'Arcueil Silic 137
94523 Rungis Cedex
Tel. (01) 687 23 13
Sté Commerciale Toutélectric (Main Office)
15-17, Boulevard Bonrepos - 31008 Toulouse
Tel. (61) 62 11 33
Sté. Commerciale Toutélectric
80-83, quai des Queyries 33100 Bordeaux
Tel. (56) 86 50 31

GERMANY
Alfred Neye Enatechnik GmbH (Ab 1.1.1979)
Schillerstrasse 14 D-2085 Quickborn/Hamburg
Tel. (04) 106 61 21
Distron oHG
Beheimstrasse 3
1000 Berlin 10
Tel. (030) 342 10 41/45
EBV Elektronik Vertriebs-GmbH (Main Office)
Gabriel-Max-Strasse 72
8000 Munchen 90
Tel. (089) 64 40 55
EBV Elektronik Vertriebs GmbH
In der Meineworth 9
3006 Burgwebel 1 / Hannover
Tel. (051) 39 45 70
EBV Elektronik Vertriebs-GmbH
Oststrasse 129
4000 Dusseldorf
Tel. (0211) 8 48 46
EBV Elektronik Vertriebs GmbH
Mylnusstrasse 54
6000 Frankfurt 1
Tel. (06111 72 04 16
EBV Elektronik Vertriebs-GmbH
Alexanderstrasse 63
7000 Stuttgart 1
Tel. (0711) 24 74 81
Jermyn GmbH
Postfach 1180
6277 Camberg
Tel. (06434) 60 05
Mütron Müller & Co. KG
Bornstrasse 22
2800 Bremen
Tel. (0421) 31 04 85
RTG, E. Springorum GmbH + Co. (Main Office)
Postfach 426
46 Dortmund 1
Tel. (02311 5 49 51
RTG, E. Springorum GmbH + Co.
Friedrich-Ebert Damm 112
2000 Hamburg 70
Tel. (040) 693 70 61 62
RTG, E. Springorum GmbH + Co.
Ungererstrasse 43
8000 Munchen 40
Tel. (089) 36 65 00
RTG, E. Springorum GmbH + Co.
Reutlingerstrasse 87
7000 Stuttgart-Degerloch
Tel. (0711) 76 64 28
RTG, E. Springorum GmbH + Co.
Mendelssohn-Bartholdy-Strasse 16
6200 Wiesbaden
Tel. (06121) 52 73 09
SASCO Vertrieb von elektronischen Bauelementen GmbH (Main Office)
Hermann-Oberth-Strasse 16
8001 Putzbrunn b. Munchen
Tel. (089) 46 40 61/69
SASCO Vertrieb von elektronischen Bauelementen GmbH
Postfach 3066
4005 Dusseldorf-Meerbusch 3
Tel. (02150) 14 33
SASCO Vertrieb von elektronischen Bauelementen GmbH
Postfach 890214
3000 Hannover
Tel. (0511) 86 25 86
SASCO Vertrieb von elektronischen Bauelementen GmbH
Lorenzer Strasse 15
8500 Nurnberg
Tel. (0911) 20 41 52
SASCO Vertrieb von elektronischen Bauelementen GmbH
Stafflenbergstrasse 24 - 7000 Stuttgart 1
Tel. (0711) 24 45 21
SPOERLE Electronic (Ab 1.1.1979)
Otto-Hahn-Strasse 13 - 6072 Dreieich b. Frankfurt
Tel. (06103) * 3 04-1
Technoprojekt (Main Office)
Heinrich-Ebner-Strasse 13
7000 Stuttgart - Bad Cannstatt
Tel. (0711) 56 17 12
Technoprojekt
Ostring 150 - 6231 Schwalbach/Ts
Tel. (06196) 8 21 00

GREECE
Macedonian Electronics Ltd.
Charilaou - P.O. Box 240 - Thessaloniki
Tel. 30 68 00
Macedonian Electronics Ltd.
Lloyd George 10 - Athens
Tel. (21) 360 95 71

HOLLAND
B.V. Diode
Hollantlaan 22 Utrecht
Tel. (030) 88 42 14
Manudax Nederland B.V.
Meerstraat 7
5473 ZG Heeswijk (N.B.) P.O. Box 25
Tel. (41) 39 12 52

HUNGARY
Interag Co., Ltd.
XIII Rajk Laszlo u. 11 P.O. Box 184
1330 Budapest
Tel. (1) 32 93 40

IRAN
Milcom LTD, Motorola Building
Nuloo Street, Vanak Square Teheran
Tel. 66 12 14 15

ITALY
Celdis Italiana S.p.A. (Main Office)
Via F.lli Gracchi 36 20092 Cinisello Balsamo (MI)
Tel. (02) 612 00 41-2-3-4-5
Celdis Italiana S.p.A.
Via Lorenzo il Magnifico 109 00162 Roma
Tel. (06) 42 38 55
Celdis Italiana S.p.A.
Via Turati, 33 40055 Castenaso (Bologna)
Tel. (051) 78 80 78
Celdis Italiana S.p.A.
Via Mombarcaro 98 10136 Torino
Tel. (11) 35 93 12
Celdis Italiana S.p.A.
Via Ognissanti, 83 - 35100 Padova
Tel. (049) 2 68 02
Cramer Italia S.p.A. (Main Office)
Via Cristoforo Colombo 134 - 00147 Roma
Tel. (06) 51 79 81-2-3-4-5-6-7
Cramer Italia S.p.A.
Via S. Simpliciano 2 - 20121 Milano
Tel. (02) 80 93 26
Cramer Italia S.p.A.
Via Umberto 1 59 - 35100 Padova
Tel. (049) 2 53 74
Cramer Italia S.p.A.
Via Malta 5 - 40135 Bologna
Tel. (051) 42 28 90
Cramer Italia S.p.A.
Corso Traiano 28/15 - 10135 Torino
Tel. (011) 619 20 62
Silverstar Ltd. S.p.A.
Via dei Gracchi 20 20146 Milano
Tel. (02) 49 96
Silverstar Ltd. S.p.A.
Via Parsiello 30 - 00198 Roma
Tel. (06) 844 88 41
Silverstar Ltd. S.p.A.
Piazza Adriano 9 10139 Torino
Tel. (011) 44 32 75/6 44 23 21

NIGERIA
Beckron International
14, Alhaji Bashorun Street
SW Ikoyi P.O. Box 1896 Lagos
Tel. 5 66 29

NORWAY
Ola Tandberg Elektro A/S
Skedsmogatan 25 Oslo 6
Tel. (02) 19 70 30

POLAND
PHZ Transpol S.A. (Intraco Building)
Ul. Stawki 2 - 00-950 Warsaw 1
Tel. (004822) 39 50 79

PORTUGAL
Equipamentos de Laboratorio LDA
Rua Pedro Nunes 47 Lisbon 1
Tel. 97 02 51

SOUTH AFRICA
L'Electron
704 Main Pretoria Road, Wynberg Tvl.
P.O. Box 10544, Johannesburg 2000
Tel. 40 62 96

SPAIN
Hispano Electronica S.A. (Main Office)
Polígono Industrial "Urtinsa"
Apartado de Correos 48 Alcorcón (Madrid)
Tel. (01) 619 41 08
Hispano Electronica S.A.
Figols, 27-29
Barcelona 14 Tel. 250 05 22 23

SWEDEN
Interelko AB.
Sandsborgsvagen 50
12233 Enskede Tel. (08) 49 25 05
AB Gösta Backstrom
Alstromergatan 22 Box 12009
10221 Stockholm
Tel. (08) 54 10 80

SWITZERLAND
Elbatex AG
Alb. Zwyssig-Strasse 28 5430 Wettingen
Tel. (056) 26 56 41
Omni Ray AG
Dufourstrasse 56 8008 Zurich
Tel. (01) 34 07 66

TURKEY
ERA Elektronik Sanyi Ve Ticaret A.S.
Ankara Irtihat Burosu
Gazi Mustapha Kemal Bul. 12
KAT4D79 Yenisehir/Ankara Tel. 25 49 33

UNITED KINGDOM
A.M. Lock & Co. Ltd
Neville Street, Middleton Road
Oldham, Lancs OL9 6LF
Tel. (061) 652 04 31
Celdis Ltd.
37-39 Loverock Road
Reading, Berks, RG3, 1ED
Tel. (0734) 585 171
Cramer Components Ltd
Hawke House Green Street
Sunbury on Thames, Middlesex, England
Tel. (9327) 8 55 77
Crellon Electronics Ltd.
380, Bath Road
Slough, Berks SL1 6JE
Tel. (06286) 6 36 11
ITT Electronic Services
Edinburgh Way
Harlow, Essex CM20 (2DF)
Tel. Harlow (0279) 26 777
Jermyn Industries
Vestry Estate Sevenoaks, Kent
Tel. (732) 5 11 74
Macro-Marketing Ltd.
396, Bath Road
Slough, Berks SL1 6JD
Tel. (06286) 630 11

YUGOSLAVIA
Elektrotehna Ljubljana
Export-Import
Titova 51 P.O. Box 34-1
61000 Ljubljana
Tel. (61) 32 02 41
Elektrotehna Ljubljana
Filiala Beograd
Marsala Tita 6/1
11000 Beograd
Tel. (011) 69 69 24

EUROPEAN SEMICONDUCTOR FACTORIES

FRANCE
Motorola Semiconducteurs S.A.
Canto Laouzetto - Le Mirail
31023 Toulouse CEDEX
Tel. (61) 40 11 88

GERMANY
Motorola GmbH
Munchner Strasse 18
8043 Unterfohring
Tel. (089) 92 481

UNITED KINGDOM
Motorola Semiconductors Ltd.
Colvilles Road, Kelvin Estate
East Kilbride/Glasgow (Scotland)
Tel. (3552) 39 101

This was brought to you

from the archives of

http://retro-commodore.eu